

# Compressed bitmap indexes: beyond unions and intersections

Owen Kaser<sup>1\*</sup> and Daniel Lemire<sup>2</sup>

<sup>1</sup>*Dept. of CSAS, University of New Brunswick, Saint John, NB, Canada*

<sup>2</sup>*LICEF, TELUQ, Université du Québec, Montreal, QC, Canada*

## SUMMARY

Compressed bitmap indexes are used to speed up simple aggregate queries in databases. Indeed, set operations like intersections, unions and complements can be represented as logical operations (AND, OR, NOT) that are ideally suited for bitmaps. However, it is less obvious how to apply bitmaps to more advanced queries. For example, we might seek products in a store that meet some, but maybe not all, criteria. Such threshold queries generalize intersections and unions; they are often used in information-retrieval and data-mining applications. We introduce new algorithms that are sometimes two orders of magnitude faster than a naïve approach. Our work shows that bitmap indexes are more broadly applicable than is commonly believed.

**KEY WORDS:** T-overlap queries; compressed bitmaps; threshold functions; symmetric functions; opt-threshold queries

## 1. INTRODUCTION

There are many applications for bitmap indexes, from conventional databases (e.g., Oracle [1]) all the way to information retrieval [2] and column stores [3]. They are used in data-warehouse platforms such as Apache Hive, LucidDB [4] and Sybase IQ [5].

We are primarily motivated by the application of bitmap indexes to common databases (i.e., row stores). In this case, it has long been established that bitmap indexes can speed up several queries, e.g., joins [6], as well as intersections and unions (e.g., `SELECT * WHERE A=1 AND B=2`).

Databases are commonly used for data mining and machine learning. An algorithm could seek to identify all movies that are “similar” to a target movie, or all customers that “almost” fit a given profile. Such queries need neither an intersection nor a union, but something in-between: a threshold function where only some of the criteria need to be satisfied. We aim to show that such queries (specifically *Many-Criteria* queries and *Similarity* queries, see § 4) can be answered efficiently using bitmap indexes. Because the result of the query is itself a bitmap, we can then further process it using the standard operations permitted on bitmaps (OR, AND, XOR, NOT) to efficiently answer more complicated queries.

Of course, the set of basic operations supported by bitmap indexes may be sufficient to synthesize any required function. However, the efficiency of such approaches is unknown. To our knowledge, the efficient computation of threshold functions over bitmaps has never been investigated.

This paper considers several algorithms for threshold functions over compressed bitmap indexes (see Table I). Some of these algorithms are novel (LOOPED, SSUM, w2CTI), whereas other algorithms are adaptations of known algorithms that had operated over sorted integer lists

---

\*Correspondence to: Owen Kaser, Dept. of CSAS, University of New Brunswick, 100 Tucker Park Rd, Saint John, NB E2L 4L1 Canada. email: owen@unbsj.ca

Table I. Algorithms considered in this paper.

Algorithm	Source	Section
SCANCOUNT	[8]	§ 6.1
W2CTI	novel	§ 6.1.1
MGOPT	[9, 10]	§ 6.2
DSK	[8]	§ 6.2
SSUM	novel	§ 6.3.1
LOOPED	novel	§ 6.4
RBMRG	modified from [11]	§ 6.5

(SCANCOUNT, MGOPT, DSK). (A companion report [7] considers additional algorithms that do not perform as well, and also considers the use of uncompressed bitmaps.) The theoretical analyses of these alternatives, summarized in Table III<sup>†</sup>, suggests that there would be no single best algorithm for all cases, as the algorithms’ running times depend on different factors. Experiments described in § 7 confirm this conclusion. Thus one of our contributions is a set of rules for automatically choosing algorithms.

Our work is organized as follows. In § 2, we formalize the problem. In § 3, we present some background material and related work. In § 4, we present the queries over database tables that we use for benchmarking. In § 5, we begin our experimental report by showing that using a bitmap index, albeit naïvely, is better than a full table scan: the indexed version is anywhere from 1.1 to 6 times faster. In § 6, we present our various algorithms. Finally, in § 7 we assess them experimentally and show that one can do significantly better than a naïve approach: up to  $740\times$  better, in one case. Over a large workload that we constructed, we could more than triple performance.

## 2. FORMULATION

We take  $N$  *sorted* sets over a universe having  $r$  distinct values. For our purposes, we represent sets as bitmaps using  $r$  bits. For example, if  $N = 2$  and  $r = 8$ , we might have the sets  $\{1, 4, 5\}$  and  $\{4, 5, 7\}$  of integers in  $[0, 8)$  represented using the bitmaps 00110010 and 10110000 (see § 3.1). The notation we use throughout is described in Table II.

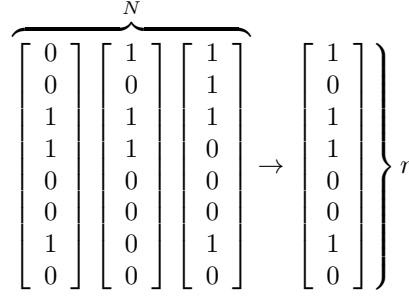
The sum of the cardinalities of the  $N$  sets is  $B$ : in our example  $B = 3 + 3 = 6$ . The cardinality of a set is also given by the *number of 1s* in the corresponding bitmap. (By extension, the cardinality of a bitmap is the *number of 1s* it contains.) Therefore, the value  $B$  is also the total number of 1s in all bitmaps. We apply a threshold  $T$  ( $1 \leq T \leq N$ ), seeking those elements that occur in at least  $T$  sets (see Fig. 1). Because the cases  $T = 1$  and  $T = N$  correspond to intersections and unions, which are well understood, we assume that  $2 \leq T \leq N - 1$ . These queries are often called  $T$ -overlap [12, 13],  $T$ -occurrence [8, 14] or  $T$ -threshold [10, 15] queries.

We can map a  $T$ -overlap query to a query over bitmaps using a Boolean threshold function: given  $N$  bits, the  $T$ -threshold function  $\vartheta(T, \{b_1, \dots, b_N\})$  returns true if at least  $T$  bits are true; it returns false otherwise. For example, given  $T = N$ , such a function would just be a logical conjunction (AND) and given  $T = 1$ , it would be a logical disjunction (OR). That is, we have  $\vartheta(N, \{b_1, \dots, b_N\}) = b_1 \wedge \dots \wedge b_N$  and  $\vartheta(1, \{b_1, \dots, b_N\}) = b_1 \vee \dots \vee b_N$ .

A (unary) bitmap index over a table has as many bitmaps as there are distinct attribute values (see Fig. 2). Each attribute value (say value  $v$  of attribute  $a$ ) has a bitmap that encodes the set of row IDs that satisfy the predicate  $a = v$ . A  $T$ -overlap query seeks row IDs that satisfy at least  $T$  of  $N$  predicates. Since each predicate is encoded as a bitmap, we need to compute a bitwise threshold function over the  $N$  chosen bitmaps.

Threshold functions are a subset of the *symmetric Boolean functions* (see § 3.2). They include the majority function: given  $N$  bits, the majority function returns true when  $1 + \lfloor N/2 \rfloor$  or more

<sup>†</sup>The notation used throughout can be found in Table II.

Figure 1. Solution to a threshold query with  $T = 2$  over  $N = 3$  bitmaps.

Name	City		Montreal	Toronto	Paris
John	Montreal		1	0	0
Peter	Montreal		1	0	0
Jack	Toronto		0	1	0
Jack	Toronto	→	0	1	0
Jill	Toronto		0	1	0
Lucy	Paris		0	0	1
Mary	Toronto		0	1	0

Figure 2. Bitmap index of the attribute City.

Table II. Notation used in analyses.

Symbol	Meaning
$B_i$	$i^{\text{th}}$ bitmap
$ B_i $	number of 1s in $i^{\text{th}}$ bitmap
$B_i[j]$	value of the $j^{\text{th}}$ bit in the $i^{\text{th}}$ bitmap
$B$	$\sum_i  B_i $
$B'$	number of 1s <b>not</b> in the $T - 1$ largest bitmaps
EWASIZE	storage size in bytes of a collection of compressed bitmaps
$N$	Number of bitmaps in the query
$r$	length of bitmaps (largest index covered)
RUNCOUNT	number of runs of 0s and 1s in a collection of bitmaps
$T$	Minimum threshold
$\vartheta(T, \{b_1, \dots, b_N\})$	threshold function over bits $b_i$
$W$	machine word size

bits are true, and it returns false otherwise. We can compute the majority function as any other threshold function. Other potentially useful generalizations include setting up a maximum (no more than  $T$  bits are set) or setting a range (the number of set bits is in  $[T_1, T_2]$ ). They can be rewritten in terms of threshold functions: e.g., we can determine whether at most  $T$  bits are set by evaluating  $\vartheta(N - T, \{\neg b_1, \dots, \neg b_N\})$ . We do not consider such generalizations further.

We denote the processor's native word length as  $W$  (typically<sup>‡</sup>  $W = 64$ ). An uncompressed bitmap will have  $\lceil r/W \rceil$  words; given  $N$  bitmaps, there are  $N \lceil r/W \rceil$  words. To simplify, we assume  $\log N < W < r$  as well<sup>§</sup> as  $\log r \leq W$ , which would typically be the case in the applications we envision. Also, we assume that  $B \geq N$ , which would be true if there is no bitmap containing only 0s: such *empty* bitmaps could be virtually deleted without harm.

<sup>‡</sup>Common 64-bit PCs have SIMD instructions that work over 128-bit (SSE and AVX), 256-bit (AVX2) or 512-bit (AVX-512) vectors. These instructions might be used automatically by compilers and interpreters. Other general purpose processors in embedded or mobile devices sometimes have a 32-bit word size.

<sup>§</sup>In this paper,  $\log n$  means  $\log_2 n$ .

Table III. Time and memory complexity of threshold algorithms over RLE-compressed bitmap indexes.

Algorithm	Time complexity	Memory	Comment
SCANCOUNT	$O(r + B)$	$O(r)$	Efficient access pattern
w2CTI	$O(B(N - T))$	$O(B)$	
MGOPT	$O(B'(\log(N - T) + T) + B - B')$	$O(N)$	Pruning can reduce $B$
DSK	$O(B'(\log(N - T) + T) + B - B')$	$O(N)$	Pruning can reduce $B'$ & $B$
SSUM	$O(Nr/W)$	$O(Nr/W)$	Note <sup>1</sup>
LOOPED	$O(NT r/W)$	$O(T r/W)$	Note <sup>2</sup>
RBMrg	$O(\text{RUNCOUNT} \times \log N)$	$O(N)$	

<sup>1</sup> Approximately  $5N$  basic bitmap operations are used, producing temporary results. An  $O(Nr/W)$  bound ignores any benefits of compression for storage or processing. With data-flow analysis (see § 6.3.2), the space complexity would typically be much lower; temporaries are only retained until their last use.

<sup>2</sup> Fewer than  $2NT$  basic bitmap operations are used, and  $T$  temporary bitmaps are used. The bounds shown ignore compression's benefits.

Our focus is on algorithms that run in main memory; we assume that the  $N$  bitmaps involved in the threshold query have already been read into main memory. Our memory bounds in Table III are based on the *additional* working memory required, not including the input and output.

**A lower bound (and beating it):** Towards a lower bound for the problem, note that if the output indicates that  $X$  entries meet the threshold, at least  $TX$  1s have been observed in the input. If each such observation triggers  $\Omega(1)$  work (as it does with SCANCOUNT (§ 6.1), when a counter is incremented), this implies an  $\Omega(TX)$  lower bound. Barbay and Kenyon [10] have established a data-dependent lower bound for the problem, assuming the data is presented in sorted arrays and using a model where comparisons are the only allowed operations on array elements. However, both bounds leave open the possibility of using parallelism. One such approach, parallelization of SCANCOUNT on GPUs, is described by Li et al. [12]. We can use bit-level parallelism (readily available in bitmap inputs) to process several events per machine operation. (See § 6.3 and § 6.4.) The bounds also leave open the possibility of using Run Length Encoding (RLE), whereby many consecutive events can be succinctly represented and processed together. Our compressed bitmap inputs are suitable for such an approach: see § 6.5.

### 3. BACKGROUND AND RELATED WORK

We review some key concepts on compressed bitmaps, Boolean circuits, and Boolean functions—especially symmetric and threshold functions.

#### 3.1. Bitmaps

We consider compressed and uncompressed bitmaps. The *density* of a bitmap is the fraction of its bits that are 1s. A bitmap with low density is *sparse*, and such bitmaps arise in many applications. A bitmap with density closer to 1 (perhaps 5 % or more) is *dense*.

**Uncompressed Bitmaps** An uncompressed bitmap represents a sorted set  $S$  over  $\{0, 1, \dots, r - 1\}$  using  $\lfloor (r + W - 1)/W \rfloor$  consecutive words. They are also called bitsets, bit vectors or bit sets. The  $W$  bits in the first word record which values in  $[0, W - 1]$  are present in  $S$ . The bits in the second word record the values in  $[W, 2W - 1]$  that are in  $S$ , and so forth. For example, the set  $\{1, 2, 7, 9\}$  is represented as 10000110 00000010 with  $W = 8$ . Its density is  $\frac{4}{16}$ . The exact mapping between integers and the bits within a word is unimportant, as long as it is always consistent (e.g.,  $\{1, 2, 7\}$  could be written as 10000110 or 01100001). The number of 1s is always equal to the cardinality of the set.

Uncompressed bitmaps have the advantages of a fixed size (updates do not change the size). and an efficient membership test. However, if  $r$  is large, the bitmap occupies many words and uses much memory—even when representing a small set ( $B \ll r$ ).

**Compressed Bitmaps** In a bitmap, there are runs of consecutive 0s and runs of consecutive 1s. The number of such runs is called the **RUNCOUNT** of a bitmap, or of a collection of bitmaps [16]. For example, in the bitmap index illustrated by Fig. 2, there are  $2 + 4 + 3 = 9$  runs. In the unary bitmap index of an attribute containing  $N$  distinct attribute values, given that there are  $r$  rows, the number of runs must be between  $3N - 2$  and  $2r + N - 2$ . Correspondingly, for  $r \gg N$ , the average length of the runs is between  $\approx r/3$  and  $\approx N/2$ . In many situations where bitmaps are generated, we expect to find many long runs (e.g., with length greater than  $W$ ).

Though there are alternatives [17], the most popular compression techniques are based on the (word-aligned) RLE compression model inherited from Oracle (BBC [1]): WAH [18], Concise [19], EWAH [11], COMPAX [20], VAL-WAH [21], among others. The  $r$  bits of the bitmap are partitioned into sequences of  $W'$  consecutive bits, where  $W' \approx W$  depends on the technique used; for EWAH,  $W' = W$ ; for WAH,  $W' = W - 1$ . When such a sequence contains only 1s or only 0s, it is a *fill* word, otherwise it is a *dirty* word. For example, using  $W' = 8$ , the uncompressed bitmap 000000001010000 contains two words, a fill word (00000000) and a dirty word (01010000). Techniques such as BBC, WAH or EWAH typically use special marker words to compress long sequences of identical fill words. When accessing these formats, it may be necessary to read every compressed word to determine whether it indicates a sequence of fill words, or a dirty word. The EWAH format [11] supports a limited form of skipping because it uses marker words not only to mark the length of the sequences of fill words, but it also uses these markers to indicate the lengths of the sequences of consecutive dirty words. Because of this feature, one can skip sequences of dirty words when using EWAH.

Though there are many good compressed formats to choose from, we have picked EWAH. In a benchmark between various formats where the authors used our implementation (the JavaEWAH library [22]), Guzun et al. [21] found that “Although EWAH does not compress well, (...) it offers the best query time for all distributions.” Moreover, EWAH is used in a major data database system (Apache Hive). We refer the reader to previous work for the exact format specification [11].

Compressed bitmaps are often appropriate for storing sets that cannot be efficiently handled by uncompressed bitmaps. For instance, consider the bitmap consisting of a million 0s followed by a million 1s. This data has two runs (**RUNCOUNT** = 2) but a million 1s. It can be stored using EWAH in only a few words.

However, some RLE compressed bitmaps are not efficient for storing extremely sparse data that does not have dense clusters. For instance, consider EWAH: sparse data with very long runs of 0s between elements will result in a marker word and a dirty word for each 1 bit. Because EWAH uses 64-bit words by default, we would use 128 bits per element. This would be less efficient than explicitly listing the set elements (e.g., 32 bits) by a factor of 4. Observe, however, that using (compressed) bitmaps for such sets is likely inefficient in any case: bitmaps are efficient due to bit-level parallelism when there are many words containing a mix of 1s and 0s.

Software libraries for compressed bitmaps will typically include an assortment of basic Boolean operations that operate directly on the compressed bitmaps. One would expect to find operations for AND, OR, and often one finds XOR, ANDNOT, and NOT. EWAH, like most other RLE-based formats, allows the operations AND, OR, XOR, ANDNOT between two compressed bitmaps ( $B_1$  and  $B_2$ ) to execute in time  $O(\text{EWAHSize}(B_1) + \text{EWAHSize}(B_2))$ . Moreover, the output of such an aggregate has compressed size bounded by the size of the input ( $\text{EWAHSize}(B_1) + \text{EWAHSize}(B_2)$ ). (For AND, the output is bounded by  $\min(\text{EWAHSize}(B_1), \text{EWAHSize}(B_2))$ .)

Some libraries support only binary operations, whereas others support *wide* queries: for instance, a wide AND would allow us to intersect four bitmaps in a single operation, rather than having to AND bitmaps together pairwise. Explicit support for wide operations can allow for better performance [23]. Threshold functions are wide queries when  $N > 2$ .

Our complexity analysis (Table III) assumes that we can iterate over the 1s in a compressed bitmap in  $\Theta(1)$  time. We can indeed iterate over the 1s in a compressed EWAH bitmap quickly. Runs of fill words are not problematic: e.g., 64-bit EWAH uses 32-bit counters for the length of such runs, so runs of up to  $2^{32} \times 2^W$  identical bits can be marked with a single marker word. Moreover, we can also extract 1s from dirty words quickly. In Java, we can use the `Long.numberOfTrailingZeros` function and a simple loop: this function is commonly compiled to efficient machine instructions by the JVM (e.g., `bsr` on Intel processors).

### 3.2. Boolean Functions and Circuits

A Boolean function is a function of the form  $f : \{0, 1\}^k \rightarrow \{0, 1\}$ . For relevant background on Boolean functions, see Knuth [24]. A Boolean circuit over some basis (e.g., AND, OR, NOT) is a directed acyclic graph where each vertex is either a basis function or an input, and where some of the vertices are outputs. Boolean functions can be computed by Boolean circuits. As discussed in § 2, some Boolean functions are *symmetric*. These functions are unchanged under any permutation of their inputs. I.e., a symmetric function is completely determined if one knows the number of 1s (the Hamming weight) in its inputs. An example symmetric function outputs 0  $\iff$  the Hamming weight of its inputs is a multiple of 2: this is the XOR function.

### 3.3. Threshold Functions

Threshold functions (in the guise of  $T$ -overlap queries) have been used for approximate searching. Specifically, Sarawagi and Kirpal [9] show how to avoid unnecessary and expensive pairwise distance computations (such as edit-distance computations) by using threshold functions to screen out items that cannot be approximate matches. Their observation was that strings  $s_1$  and  $s_2$  must have many ( $T$ )  $q$ -grams in common, if they have a chance of being approximate matches to one another. Given  $s_1$  and seeking suitable  $s_2$  values, we take the set of  $q$ -grams of  $s_1$ . Each  $q$ -gram is associated with a set of the words (more specifically, with their row IDs) that contain that  $q$ -gram at least once. Taking these  $N$  sets, we use a threshold function to determine values  $s_2$  that can be compared more carefully against  $s_1$ . Using  $q$ -grams, Sarawagi and Kirpal showed that  $T = |s_1| + q - 1 - kq$  will not discard any string that might be within edit distance  $k$  of  $s_1$ . In applications where  $k$  and  $q$  are small but the strings are long, this will create queries where  $T \approx N$ . (Similar formulae are known for Jaccard, cosine and dice similarities [8, 9].)

Closely related to  $T$ -overlap queries, we have Opt-threshold queries [10, 25]. In these queries,  $T$  is not specified: the algorithm is responsible for choosing the largest threshold value that leads to a non-empty result. We could further generalize such queries by asking for the largest value  $T$  such that the result of the  $T$ -overlap query contains at least  $K$  elements.

Knuth [24, 7.1.1&7.1.2] discusses threshold functions and symmetric functions, as well as techniques by which other symmetric functions can be synthesized from threshold functions.

## 4. ADVANCED QUERIES

To obtain results that correspond to a practical applications of bitmap indexes, we focus on using threshold functions over bitmap indexes to answer two different types of queries, *Many-Criteria* queries and *Similarity* queries.

**Many-Criteria Queries:** The first type of query has a set of criteria, and we are seeking those records that meet some minimum number of the criteria, but perhaps not all. E.g., consider a query that might be typical of some human-resources system (in pseudo-SQL).

```
SELECT * FROM table WHERE Gender="F" AND City="Montreal"
AND experience>=24 AND education>=college;
```

If it corresponds to an application where we filter job candidates, maybe applying all constraints at once could lead to a small (or empty) result set. Or maybe we want to include exceptional candidates

who fail to satisfy a few conditions. So we are willing to relax the constraint somewhat, by maybe requiring that only three of the constraints hold, as in the following example.

```
SELECT * FROM table WHERE
  CASE WHEN Gender="F" THEN 1 ELSE 0 END
+ CASE WHEN City="Montreal" THEN 1 ELSE 0 END
+ CASE WHEN experience>=2 THEN 1 ELSE 0 END
+ CASE WHEN education >= college THEN 1 ELSE 0 END
>= 3;
```

More complex queries are possible: instead of requiring a specific attribute value—as in `City="Montreal"`—we could require one of several possible attribute values—as in `(City="Montreal" or City="Vancouver")`. However, since attributes can only take one value in a relational system, we can transform the more complex query back into our simple form.

```
SELECT * FROM table WHERE
  CASE WHEN Gender="F" THEN 1 ELSE 0 END
+ CASE WHEN City="Montreal" THEN 1 ELSE 0 END
+ CASE WHEN City="Vancouver" THEN 1 ELSE 0 END
+ CASE WHEN experience>=2 THEN 1 ELSE 0 END
+ CASE WHEN education >= college THEN 1 ELSE 0 END
>= 3;
```

**Similarity Queries:** The second type of query presents a prototypical item. We determine the criteria that this item meets, and then seek all items that meet (at least)  $T$  of these criteria. For example, if a user liked a given movie, he might be interested in other similar movies (e.g., same director, or same studio, or same leading star, or same date of release). As part of a recommender system, we might be interested in identifying quickly all movies satisfying at least  $T$  of these criteria. This might be viewed as setting a threshold on the Hamming distance between tuples.

Once the criteria have been defined, SQL can handle the rest of the query, as in the following example.

```
SELECT * FROM movies WHERE
  CASE WHEN Lead="Sean Penn" THEN 1 ELSE 0 END +
  CASE WHEN Studio="Paramount" THEN 1 ELSE 0 END +
  CASE WHEN Director="Ridley Scott" THEN 1 ELSE 0 END
> 2;
```

Critchley [26] proposes an alternative SQL-only solution using joins and SQL aggregation; his approach resembles that of `WSORT`, a less-than-ideal algorithm discussed in our companion report. However, his approach works in external memory.

Similarity queries have been used with approximate string matching [8, 9]. In this case, items are small chunks of text, and the occurrence of a particular 3-gram (a sequence of 3 consecutive letters) is a criterion. In that previous work, an index maps each 3-gram to a sorted list of integers that specify the chunks of text containing it. Ferro et al. [27] solve a similar problem, but instead use a bitmap for each 2-gram. Montaneri and Puglisi [28] extend the bitmap approach to detect near-duplicate documents arriving over time. In related work, Tellez et al. [25] use such a threshold approach to accelerate nearest neighbor search by pruning the candidate list.

A generalization of a Similarity query presents *several* prototypical items, then determines the criteria met by at least one of them. We then proceed as before, finding all items in the database that meet at least  $T$  of the criteria. If there are  $n$  prototypes, we have a “Similarity( $n$ )” query.

Assuming one has a bitmap index, can one answer Many-Criteria and Similarity queries better than using the row-scan that would be done by a typical database engine? One of our contributions is to show that it is indeed the case. In § 5, we show that a simple bitmap-based algorithm (`SCANCOUNT`, see § 6.1) is able to outperform a row scan (e.g., by a factor of 6). Then in § 7 we show that other bitmap-based algorithms can outperform this simple approach (`SCANCOUNT`), sometimes by hundreds of times.

---

**Algorithm 1** Row-scanning approach over a row store.

---

**Require:** A table with  $D$  attributes. A set  $\kappa$  of  $N \leq D$  attributes, and for each such attribute a desired value. Some threshold  $T$ .

```

1: Create an initially empty set  $s$ 
2: for each row in the table do
3:   counter  $c \leftarrow 0$ 
4:   for each attribute  $k$  in  $\kappa$  do
5:     if attribute  $k$  of the row has the desired value then
6:       increment  $c$ 
7:   if  $c \geq T$  then
8:     add the row (via a reference to it) to  $s$ 
9: return the set of matching rows,  $s$ 

```

---

Table IV. Total time (ms) required for queries in our workload.  
Top: Many-Criteria query. Bottom: Similarity query.

	CensusIncome	Weather	TWEED
EWAH SCANCOUNT	109	201	6
Row Scan (no index)	487	1212	23
Row Scan/SCANCOUNT(%)	450	600	380
EWAH SCANCOUNT	327	508	20
Row Scan (no index)	557	1344	22
Row Scan/SCANCOUNT(%)	170	260	110

## 5. AN INDEX IS BETTER THAN NO INDEX

Could a simple T-occurrence query can be more effectively answered without using a bitmap index? Before continuing our investigation with various novel algorithms, we want to establish that bitmap indexes can accelerate some T-occurrence queries. Our purpose is merely motivational: detailed experiments, including a description of our queries and datasets is given in § 7.

As a reference, we use a full table scan (see Algorithm 1). To test out the basic usefulness of a bitmap index, we use a simple algorithm (SCANCOUNT, see § 6.1 for details): we create an array of  $r$  counters initialized to zero. Then the bits of each bitmap are scanned in sequence, one bitmap at a time. When a 1-bit is found, the corresponding counter is incremented. The algorithm concludes with a full scan of the all counters.

We made 30 trials, on each of the datasets CensusIncome, Weather and TWEED. These are described in § 7.2 and have 42, 19 and 53 attributes, respectively. We randomly chose one value per attribute and randomly chose a threshold between 1 and the number of attributes, exclusively. This query corresponds to a Many-Criteria query. Table IV shows that using an EWAH index for this query was 4–6 times faster than scanning the table. The advantage persisted, but was smaller, when we did a Similarity query against a randomly chosen row. It is reassuring that a bitmap index using SCANCOUNT answered our queries faster than they would be computed from the base table. It remains to determine whether we can surpass SCANCOUNT. Section 7 shows that three algorithms can run at least  $500\times$  faster than SCANCOUNT on certain queries, although speedups of  $3\times$  to  $5\times$  seem more typical.

## 6. APPROACHES FOR THRESHOLD FUNCTIONS

We next present several different approaches to computing threshold functions. Several generalize to handle all symmetric functions, and several can be modified to solve Opt-threshold queries.



### 6.1. Counter-based approaches

The simple SCANCOUNT algorithm of Li et al. [8] uses an array of counters, one counter per item. The input is scanned, one bitmap at a time. If an item (as a bit set to 1) is seen in the current bitmap, its counter is incremented. A final pass over the counters can determine which items have been seen at least  $T$  times. In our case, items correspond to positions in the bitmap. If the maximum bit position is known in advance, if this position is not too large, and if one can efficiently iterate the bit positions in a bitmap, then SCANCOUNT is easily implemented. These conditions are frequently met when the bitmaps represent the sets of row IDs in a table that is not exceptionally large.

SCANCOUNT is part of a family of counter-based approaches which have the characteristic that they count the occurrences of each item. They can handle arbitrary symmetric functions, since one can provide a user-defined function mapping  $[0, N]$  to Booleans. However, some counter-based approaches can be optimized specifically to compute threshold functions (see § 6.1.1).

To analyze SCANCOUNT, note that it uses  $\Theta(r)$  counters. We assume  $N < 2^W$ , so each counter occupies a single machine word. Even if counter initialization can be avoided (see Li et al. for details) the algorithm compares each counter against  $T$ . Also, the total number of counter increments is  $B$ . Together, these imply a time complexity of  $\Theta(r + B)$  and a space complexity of  $\Theta(r)$ . Aside from the effect of  $N$  on  $B$  (on average, a linear effect), note that this algorithm does not depend on  $N$ . (Li et al. present an alternative SCANCOUNT algorithm that generates an unsorted list in  $O(B)$  time. Generating a RLE-compressed bitmap would require sorting this output, and this could be a major overhead for queries with large outputs. Similar sorting requirements are imposed by an ineffective algorithm, HASHCNT, discussed in our companion paper.)

The SCANCOUNT approach fits modern hardware well: the counters are accessed in sequence, during the  $N$  passes over them when they are incremented. Experimentally, we found that using 8-bit byte counters when  $N < 128$  usually brought a small (perhaps 15%) speed gain compared with 32-bit int counters. Perhaps more importantly, this also quarters the memory consumption of the algorithm. One can also experiment with other memory-reduction techniques: e.g., if  $T < 128$ , one could use a saturating 8-bit counter. Experimentally, we found that the gains usually were less than the losses that come from the additional conditional check required to ensure saturation. Based on our experimental results, the SCANCOUNT implementation used in § 7 switches between byte, short and int counters based on  $N$ , but does not use the saturating-count approach.

SCANCOUNT fails when the bitmaps have extreme  $r$  values. If we restrict ourselves to bitmaps that arise within a bitmap index, this implies that we have indexed a table with an extreme number of rows.

It is easy to obtain an Opt-Threshold algorithm: SCANCOUNT begins as usual and obtains the  $r$  counters.  $T$  is the maximum value in the counters, and the algorithm then returns those elements whose counters equal  $T$ . If we can reset the output, a single pass is required.

**6.1.1. Mergeable-count structures.** A common approach to computing intersections and unions of several sets is to do it two sets at a time. To generalize the idea to symmetric queries, we represent each set as an array of values coupled with an array of counters. For example, the set  $\{1, 14, 24\}$  becomes  $\{1, 14, 24\}, \{1, 1, 1\}$ , where the second array indicates the frequency of each element (respectively). If we are given a second set ( $\{14, 24, 25, 32\}$ ), we supplement it with its own array of counters  $\{1, 1, 1, 1\}$  and can then merge the two: the result is the union of two sets along with an array of counters ( $\{1, 14, 24, 25, 32\}, \{1, 2, 2, 1, 1\}$ ). From this final answer, we can deduce both the intersection and the union, as well as other symmetric operations.

Algorithm W2CTI takes this approach. Given  $N$  input bitmaps, it orders them by increasing cardinality and then merges each input, starting with the shortest, into an accumulating total. (The merge step is akin to the merge operation in the merge-sort algorithm.) A worst-case input has bitmaps of equal cardinality, each containing  $B/N$  items that are disjoint from any other input. At the  $i^{\text{th}}$  step the accumulating array of counters will have  $Bi/N$  entries and this will dominate the merge cost for the step. The total time complexity for this worst-case input is  $\Theta(\sum_{i=1}^{N-1} Bi/N) = \Theta(BN)$ . For memory use, the same input ends up growing an accumulating array of counters of size  $B$ .

Algorithm w2CTI refines this basic approach: although it ends up reading its entire input, during the merging stages it can discard elements that cannot achieve the required threshold. For instance, we can check the accumulating counters during each merge step. If there are  $i$  inputs left to merge, then any element that has not achieved a count of at least  $T - i$  can be removed from consideration (“pruned”).

In large-threshold cases, this pruning is beneficial. For instance, suppose  $T = N - \tau$  for some  $\tau \geq 1$ . Any item that has not occurred in one of the first  $\tau + 1$  bitmaps will be pruned. As these are the smallest bitmaps, they can contain no more than  $(\tau + 1)B/N$  items, and this bounds the size of the accumulator in any of the  $N$  merge operations. The total cost of the merge operations is thus in  $O(B + N(\tau + 1)B/N) = O(\tau B) = O((N - T)B)$ . However, pruning does not help much with the worst-case input, if  $T = 2$ . We cannot discard any item until the final merge is done, because the last input set could push the count (currently 1) of any accumulated item to 2, meeting the threshold. Thus, with  $T = 2$  we find a worst-case time bound of  $\Omega((N - T)B)$ .

## 6.2. *T*-occurrence algorithms for integer sets

Prior work [8, 9] has studied the case when the data is presented as sorted lists of integers rather than bitmaps. We consider the following *T*-occurrence algorithms: WHEAP [9], MGOPT [9, 10], DSK [8]. For full details of these algorithms, see the papers that introduced them. All can be viewed as modifications to the basic WHEAP approach. This approach essentially uses an  $N$ -element min-heap that contains one element per input. Using the heap, it merges the sorted input sequences. As items are removed from the heap, we count duplicates and thereby know which elements had at least  $T$  duplicates. This approach can be generalized to compute any symmetric function, but it requires that we process the 1s in each list, inserting (and then removing) the position of each into an  $N$  element min-heap. The total time cost is thus  $O(B \log N)$  for sorted lists.

The WHEAP approach has been shown to have worse performance than MGOPT or DSK [7, 8, 9] and thus is not considered further.

The remaining algorithms are also based around heaps (MGOPT and DSK), but they are designed to exploit characteristics of real data, such as skew, that allow us to skip certain input elements. In contrast with other algorithms (e.g., WHEAP, RBMRG and SCANCOUNT), MGOPT and DSK do not generalize to arbitrary symmetric functions since such functions preclude skipping any input. This is illustrated by the (wide) XOR function, whose output always depends on all input bits—knowing all but one input bit is never enough to determine the output.

**Algorithm MGOPT:** Sarawagi and Kirpal’s MGOPT algorithm [9] sets aside the largest  $T - 1$  inputs. Any item contained only in these inputs cannot meet the threshold. Then it uses an approach similar to WHEAP with threshold 1 on the smallest  $N - T + 1$  inputs. For each item found in the smallest inputs, say with count  $t$ , the algorithm checks whether at least  $T - t$  instances of the item are found in the largest  $T - 1$  inputs. The items are checked in the largest inputs in ascending sequence. If one of the largest inputs is checked for occurrence of item  $x$ , and the next check is for the occurrence of item  $y$ , we know that  $y > x$ . Items between  $x$  and  $y$  in the big input will never be needed, and can be skipped over without inspection. Whereas we use bitmaps as inputs, Sarawagi and Kirpal were using sorted lists of integers as inputs. Thus they could use a doubling/bootstrapping binary-search to find the smallest value at least as big as  $y$ , without needing to scan all values between  $x$  and  $y$ . The portions skipped have been pruned.

As noted in § 3.1, providing random access is not a standard part of a RLE-based compressed bitmap library, although it is essentially free for uncompressed bitmaps. However, with certain compressed bitmap indexes one can “fast forward”, skipping portions of the index in a limited way: the JavaEWAH library [22] uses the fact that we can skip runs of dirty words (e.g., when computing intersections).

An alternative is to convert from bitmap representation to sorted array, apply the original MGOPT algorithm, then convert the result back into a bitmap. The overheads of such an approach are usually unattractive [7].

To bound the running time, we can distinguish the  $B - B'$  1s in the  $T - 1$  largest bitmaps from the  $B'$  1s in the remaining  $N - T + 1$  bitmaps. A heap of size  $O(N - T + 1)$  is made of the  $N - T + 1$  remaining bitmaps, and  $O(B')$  items will pass through the heap, at a cost of  $O(\log(N - T + 1))$  each. As each item is removed from the heap, it will be sought in  $O(T)$  bitmaps. Because the items sought are in ascending order, the  $T - 1$  bitmaps will each be processed in a single ascending scan that handles all the searches. Each of the  $B - B'$  1s in the remaining bitmaps should cost us  $O(1)$  effort. Thus we obtain a bound of  $O(B'(\log(N - T + 1) + T) + B - B') = O(B'(\log(N - T) + T) + B - B')$  for the time complexity of MGOPT.

A similar algorithm was earlier presented by Barbay and Kenyon [10]. Any input may appear in their heap, but at any time there will be  $T - 1$  inputs that are not in the heap. Setting aside the *largest* items (as with Sarawagi and Kirpal) seems like a useful enhancement. Indeed, consider our complexity bound of  $O(B'(\log(N - T) + T) + B - B')$ : each of the  $B'$  elements has a multiplicative cost factor of  $\log(N - T) + T$  whereas each of the other  $B - B'$  elements has a cost factor of 1. This reflects the fact that the  $B'$  elements are stored in a heap whereas the  $B - B'$  elements are merely accessed sequentially. Thus we prefer to minimize  $B'$ , which is done by setting aside the largest bitmaps.

Our analysis does not take fully into account the effect of pruning, because we might be able to skip many of the  $B - B'$  1s as we search forward through the largest  $T - 1$  bitmaps. Since these are the *largest* bitmaps, if  $T$  is close to  $N$  or if the sizes (number of 1s) in the bitmaps vary widely, pruning could make a large difference. This depends on the data. Barbay and Kenyon present a detailed running-time analysis (with input as sorted integer lists) in terms of a “ $t$ -alternation” parameter for the problem instance. It matches their comparison-based lower bound for the problem in many cases, and in all cases it is within a factor of  $O(\log(N - T + 1))$  of the optimal complexity.

Barbay and Kenyon also describe how to obtain an Opt-threshold algorithm from any  $T$ -overlap algorithm by successively trying  $T = N$ ,  $T = N - 1$ , ... until a non-empty answer is obtained. Although naïve, the empty  $T$ -overlap queries have a predicable cost for MGOPT (no worse than the final query), whereas a binary search for  $T$  may make some more expensive queries.

**Algorithm DSK:** Algorithm DSK is essentially a hybrid of MGOPT and another pruning algorithm called MERGESKIP. MERGESKIP [8] is like WHEAP except that, when removing copies of an item from the heap, if there are not enough copies to meet the threshold, we remove some extra items. This is done in such a way that the extra items removed (and not subsequently re-inserted) could not possibly meet the threshold. (MERGESKIP is not described further here, because its performance is worse than DSK [7, 8]). Algorithm DSK processes the heap as in MERGESKIP, while it sets apart the largest bitmaps as in MGOPT. However, rather than following MGOPT and always setting apart the  $T - 1$  largest sets, it chooses the  $L$  largest sets where  $L$  is a tuning parameter. Li et al. determine another tuning parameter  $\mu$  experimentally, for a workload of queries against a given dataset. From  $\mu$  and the length of the longest input, Li et al. use a heuristic formula for  $L$  (see § 7.3). With a suitable  $L$ , we would not expect DSK to perform significantly worse than MGOPT.

Our running-time complexity bound for DSK is identical to that for MGOPT, and based on the same reasoning that ignores pruning. We cannot easily account for the pruning opportunities that DSK inherits from MERGESKIP and MGOPT. However, as with MGOPT, data-dependent pruning could reduce the  $B - B'$  term. As with MERGESKIP, the multiplicative  $B'$  factor can be reduced by data-dependent pruning [7].

Considering memory, note that MGOPT and DSK partition the inputs into two groups. Regardless of group, each compressed bitmap input will have an iterator constructed for it. The first group also go into a heap that accepts one element per input. Thus we end up with a memory bound of  $O(N)$ .

Table V. Number of 2-input operations for  $N = 4$  and  $N = 5$ , Knuth’s optimal solution [24], our sideways-sum adder and the LOOPED approach.

$T$	$N = 4$			$N = 5$		
	Optimal	SSUM	LOOPED	Optimal	SSUM	LOOPED
1	3	11	3	4	14	4
2	7	9	9	not given	12	12
3	7	11	13	9	14	18
4	–	–	–	10	11	22

### 6.3. Boolean synthesis

A typical bitmap implementation provides a set of basic operations, typically AND, OR, NOT, XOR and sometimes ANDNOT<sup>¶</sup>. Since one can synthesize any Boolean function using AND, OR and NOT gates in combination, any desired bitmap function can be “compiled” into a sequence of primitive operations. One major advantage is that this approach allows us to use a bitmap library as a black box, although it is crucial that the primitive operations have efficient algorithms and implementations.

Unfortunately, it is computationally infeasible to determine the fewest required primitive operations except in the simplest cases. However, there are several decades of digital-logic-design techniques that seek to implement circuits using few components [29]. Also, Knuth has used exhaustive techniques to determine the minimal-gate circuits for all symmetric functions of 4 and 5 variables [24, Figs. 9&10, 7.1.2]. His results are given in Table V and can be up to  $\approx 4$  times smaller than those we obtain from our more general constructions, discussed later in this section.

We are concerned with both running time and memory consumption. One factor that might impact memory consumption is the number of temporary bitmaps that need to be created. Can we both minimize the execution time while minimizing the number of these bitmaps? According to Knuth, all 4-input symmetric ( $N = 4$ ) functions can be evaluated in the minimum amount of memory possible for each, without increasing the number of operations [24, 7.1.2]. Thus it would seem that we could have the best of both worlds: few temporary bitmaps and few operations. Unfortunately, merely determining the minimum-memory evaluation of a circuit is NP-hard. (The related Register Sufficiency problem is NP-hard [30, A11.1.P01][31]. We have a restricted version of Register Sufficiency—we only need to compute one output, rather than an output for each root of the directed acyclic graph representing the expression. The restricted form of Register Sufficiency is also NP-hard, by reducing Register Sufficiency to it; see our companion report.) Thus, for large values of  $N$ , it might be impractical to optimally minimize the number of temporary bitmaps: we rely on heuristics.

Looking at Table V, the number of operations might seem unimpressive. Using a counter, we can compute  $\vartheta(T, \{b_1, \dots, b_N\})$  using  $N$  increments and  $N + 1$  branches. In contrast, we often require more than  $2N + 1$  bit-wise operations using our techniques (SSUM and LOOPED). Nevertheless, bit-level parallelism can make our techniques competitive. That is, even if we require more than  $2N + 1$  operations, we also compute  $W$  results at once: as long as we use fewer than  $2WN$  operations in total, we can surpass a naïve counter-based approach that uses  $\approx 2N$  operations per result.

As an illustration, consider uncompressed bitmaps using a word size  $W$ , and assume that all logic operations take unit time. Then a logic circuit with  $C$  gates can obtain a batch of  $W$  threshold results with  $C$  operations. For instance, on a 64-bit architecture, we have a circuit for  $N = 786$  that uses about 3900 gates. (The circuit is SSUM for  $N = 786$ , see § 6.3.1.) Thus 3900 bitwise operations compute 64 threshold results: we use  $3900/63 \approx 61$  operations per result. Without bit parallelism,

<sup>¶</sup>The x86 extensions SSE2 and AVX2 support AND NOT, as do several bitmap libraries (EWAH included). Specifically, Intel has the pandn and vpandn instructions; however it does not appear the standard x86 instruction set has a corresponding instruction.

Inputs			Hamming weight	Outputs	
$B_1$	$B_2$	$B_3$		$z_1$	$z_0$
0	1	1	2	1	0
0	0	1	1	0	1
1	1	1	3	1	1
1	0	0	1	0	1
	$\vdots$		$\vdots$	$\vdots$	

Figure 3. Computing the bitwise Hamming function.

at least  $786/2$  binary operations are required so that each input is used once:  $786/2$  is over 6 times larger than 61.

With a larger  $W$ , such as provided by the SSE, AVX2 and AVX-512 extensions, the advantage of this approach increases. However, this analysis ignores the fact that, given a limited number of registers, the CPU must spend some time loading bit vectors from memory and storing intermediate results there.

With RLE compressed bitmaps, the primitive operations do not all have the same computational cost. For instance, consider a bitmap  $B_1$  that consists of a single long run of 0s, bitmap  $B_2$  that contains a single long run of 1s, and a bitmap  $B_3$  that contains alternating 0s and 1s. The operations  $\text{AND}(B_1, B_3)$  and  $\text{OR}(B_2, B_3)$  are fast operations that do not require any copying or bitwise machine operations. However,  $\text{OR}(B_1, B_3)$  and  $\text{AND}(B_2, B_3)$  will be expensive operations—if  $B_3$  is mutable, then a copy must be made of it. Thus, it is simplistic to assume a cost model based on the uncompressed lengths of the bitmaps, or one that merely counts the number of bitmap operations. We might try to estimate the cost of such operations from the compressed size of the bitmaps [23] or by sampling [32]. Determining an appropriate cost estimation technique is outside our scope, however.

**6.3.1. Adding: The SSUM algorithm.** After trying several approaches described in our companion report [7], we found that our best circuit is based on summing the input bits, obtaining the Hamming weight of the input. Then a small “ $\geq$ ” circuit compares the Hamming weight against  $T$ ; the result is our SSUM algorithm.

Knuth describes a “sideways sum” circuit [24, 7.1.2] to compute the Hamming weight of a vector of bits. This circuit consists of  $O(\log N)$  levels. The first level takes a collection  $C_1$  of bits of weight 1, and as output produces a single weight-1 bit,  $z_0$ , and a collection  $C_2$  of bits of weight  $2^1$ . The Hamming weights are preserved: i.e.,  $z_0 + 2 \text{ hamming}(C_2) = \text{hamming}(C_1)$ . The second level takes  $C_2$  and produces a single weight-2 bit,  $z_1$ , and a collection  $C_3$  of bits of weight  $2^2$ . Again, Hamming weights are preserved:  $2z_1 + 4 \text{ hamming}(C_3) = 2 \text{ hamming}(C_2)$ . Note that bits  $z_1$  and  $z_0$  are the least significant bits of  $\text{hamming}(C_1)$ . As an example, see Fig. 3 where  $z_0$  and  $z_1$  are given for a case where  $N = 3$ . Subsequent levels are similar, and the Hamming weight of the  $N$  input bits is specified by the  $z$  bits. Fig. 4 shows how input bits of weight  $2^x$  are used to compute the single weight- $2^x$  bit  $z_x$  and the output bits of weight  $2^{x+1}$ . The sideways-sum circuit is composed of 5-gate full adders (each with 2 AND, 2 XOR and 1 OR gate) and 2-gate half adders (each with an AND and an XOR) used if the number of input bits is even.

For an example with  $N = 7$ , suppose the first bits in the 7 bitmaps are 1,1,1,0,1,0,1. The first level of the sideways-sum circuit produces  $z_0 = 1$  and the sequence 1,0,1 of weight-2 bits as follows: the rightmost full adder computes  $1 + 0 + 1 = 10_2$ ; the 1 becomes the rightmost weight-2 bit. The 0 is fed to the middle full adder, which adds it to the next 2 bits (1,0) of the  $N$ , getting sum  $01_2$ . The 0 becomes the middle weight-2 bit. The 1 is fed to the leftmost full adder, which adds it to the leftmost of the  $N$  bits. The resulting sum is  $11_2$ ; the least significant bit is  $z_0$ , and the most significant is the leftmost weight-2 bit. Then, the next layer of the circuit combines the weight-2 bits: a single full adder computes  $1 + 0 + 1 = 10_2$ . The 0 is  $z_1$  and the 1 is a weight-4 bit. No more layers are needed; the 1 bit becomes  $z_2$ , and thus  $z_2 z_1 z_0$  is the binary representation of our input’s Hamming weight.

We express Hamming weight using  $\lfloor \log 2N \rfloor$  bits: the minimal number of bits required to write the integer  $N$  in binary form. This  $\lfloor \log 2N \rfloor$ -bit Hamming weight is then compared to  $T$  using a  $\geq$  circuit. If  $T$  is a power of 2, it is easy to compute the threshold function from the Hamming weight using an OR function: for instance, to see if the 4-bit number  $b_3b_2b_1b_0$  is at least 2, compute  $b_1 \vee b_2 \vee b_3$ . This approach can be generalized if  $T$  is not a power of two; a *magnitude comparator* circuit [33] can determine whether the bit-string for the Hamming weight lexicographically exceeds the bit-string for  $T - 1$ . As various designs for such circuits exist, we proceed to derive and analyze our circuit to compare a quantity to a constant.

Consider two bit strings  $b_{n-1}b_{n-2} \cdots b_0$  and  $a_{n-1}a_{n-2} \cdots a_0$ , where  $n = \lfloor \log 2N \rfloor$ . The first is greater if there is some  $0 \leq j < n$  where  $b_j > a_j$  and the two bit strings have  $b_k = a_k$  for all  $k > j$ . If we define  $\text{prefix\_match}(j) = \bigwedge_{k=j+1}^{n-1} (b_k \equiv a_k)$  then  $b_{n-1}b_{n-2} \cdots b_0 > a_{n-1}a_{n-2} \cdots a_0$  can be computed as  $\bigvee_{j=0}^{n-1} \text{prefix\_match}(j) \wedge b_j \wedge \neg a_j$ . The prefix values can be computed with  $n$  XOR and  $n$  ANDNOT operations with

$$\begin{aligned} \text{prefix\_match}(n) &= 1, \\ \text{prefix\_match}(k) &= \text{prefix\_match}(k+1) \wedge \neg(b_k \oplus a_k). \end{aligned}$$

Altogether  $5n - 1$  bitwise operations (AND, ANDNOT, XOR, OR) are used to determine the truth value of the inequality  $b_{n-1}b_{n-2} \cdots b_0 > a_{n-1}a_{n-2} \cdots a_0$ .

We can do better because  $T - 1$  (whose binary representation is the second bit string,  $a_{n-1} \cdots a_0$ ) is a constant.

- First, the  $\neg a_j$  means that OR terms drop out for positions  $j$  where  $a_j = 1$ , leaving us with  $\bigvee_{j|a_j=0} \text{prefix\_match}(j) \wedge b_j$ .
- Second, the previous expression does not need  $\text{prefix\_match}$  for the trailing 1s in  $T - 1$ ; they no longer appear in our expression and there is no 0 to their right. (The  $\text{prefix\_match}$  value required for a 0 is indirectly calculated from  $\text{prefix\_match}$  values for all positions to the left of the 0.)
- Third, we can redefine  $\text{prefix\_match}(j)$  as  $\bigwedge_{\substack{j < k < n \\ \wedge a_k=1}} b_k$ . Thus, if  $T - 1 = 101100_2$ ,  $\text{prefix\_match}(2)$  no longer checks that the other bit string starts with 101. Instead, it matches 101 or 111. In the latter case, the bit string is already known to exceed  $T - 1$ .

For an example, consider  $T - 1 = a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0 = 0010100111_2$  with  $n = 10$ . We compute  $b_9 \vee b_8 \vee (\text{prefix\_match}(6) \wedge b_6) \vee (\text{prefix\_match}(4) \wedge b_4) \vee (\text{prefix\_match}(3) \wedge b_3)$ , where we have that  $\text{prefix\_match}(6) = b_7$ ,  $\text{prefix\_match}(4) = b_5 \wedge b_7 = b_5 \wedge \text{prefix\_match}(6)$  and  $\text{prefix\_match}(3) = b_5 \wedge b_7 = \text{prefix\_match}(4)$ . Thus, we finally arrive at the formula  $b_9 \vee b_8 \vee (b_6 \wedge b_7) \vee (b_4 \wedge b_5 \wedge b_7) \vee (b_3 \wedge b_5 \wedge b_7)$ . Since  $b_5 \wedge b_7$  is a shared sub-expression, we need only 8 operations in this case, which less than  $n = 10$ .

With these optimizations, computing all required  $\text{prefix\_match}$  values needs one AND for every 1 bit except for the trailing 1s and the first 1: for each of these 1 bits no gate is required. We also need a single wide OR that takes an input for each of the 0s (there must be a 0 and a 1 in the binary representation of  $T - 1$ ) in  $T - 1$ . That OR input is computed as a 2-input AND; as a minor optimization, the leading 0s in  $T - 1$  can omit the AND, because they would use a  $\text{prefix\_match}$  value that would be 1. To count operations, let  $\nu(T - 1)$  denote the Hamming weight when  $T - 1$  is written in binary, let  $\text{nto}(T - 1)$  denote the number of trailing 1s, and  $\text{nlz}(T - 1)$  denote the number of leading 0s. Therefore, the number of 0s in  $T - 1$  is  $n - \nu(T - 1)$ . Thus, computing the  $\text{prefix\_match}$  values needs  $\max(0, \nu(T - 1) - \text{nto}(T - 1) - 1)$  ANDs. We need to “OR”  $n - \nu(T - 1)$  items, requiring  $n - \nu(T - 1) - 1$  two-input OR gates. The items are computed by  $n - \nu(T - 1) - \text{nlz}(T - 1)$  AND gates. In total we use at most  $2n - \nu(T - 1) - \text{nlz}(T - 1) - \text{nto}(T - 1) - 1$  operations; i.e., between 0 (when  $T - 1 = 011 \cdots 1$ ) and  $2n - 3$  (when  $T - 1 = 100 \cdots 0$ ). The similar comparator circuit presented by Ashenden [33] can also be simplified when  $a$  values are constant. The worst case value of  $T$  gives a circuit whose size is similar to ours, but the best case is not as good.

An Opt-threshold algorithm can be obtained from SSUM with  $O(\log N)$  bitmap operations beyond those needed for SSUM. First, apply the sideways sum circuit, obtaining  $O(\log N)$  bitmaps

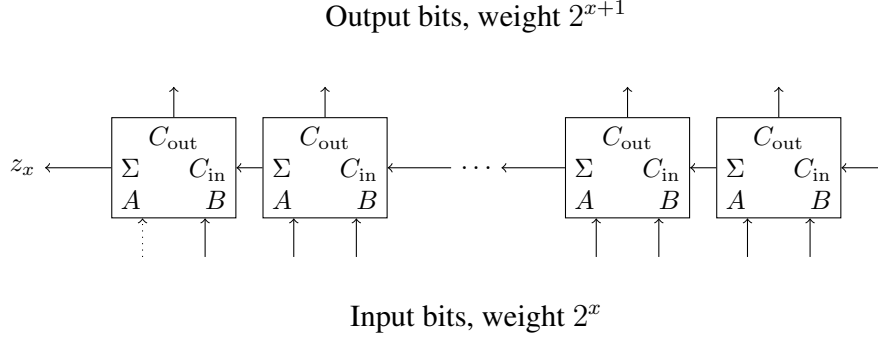


Figure 4. One layer in the sideways sum circuit, similar to that discussed by Knuth. This is not a “ripple carry” arrangement: the **sum** bit from a full adder becomes the carry-in input of the next full (or half) adder. Whether the leftmost adder is a full or a half adder depends on whether the number of input bits is even. If so, a half adder is used and the dotted input is absent.

---

**Algorithm 2** An Opt-Threshold algorithm using the bitmaps computed by SSUM.

---

**Require:** A collection of bitmaps  $B_0, B_1, \dots, B_l$  such that the binary number with digits  $B_l[j]B_{l-1}[j] \cdots B_1[j]B_0[j]$  is the occurrence count of item  $j$ .

**Ensure:**  $A[j] = 1 \iff$  item  $j$  has occurrence count  $m$ , there exists at least one item with occurrence count  $m$ , and there is no item with occurrence count greater than  $m$ .

- 1: bitmap  $A \leftarrow 111 \cdots$
  - 2: **for**  $i \leftarrow l$  **down to** 0 **do**
  - 3:   **if**  $|\text{AND}(A, B_i)| \neq 0$  **then**
  - 4:      $A \leftarrow \text{AND}(A, B_i)$
  - 5: **return**  $A$
- 

$B_l, B_{l-1}, \dots, B_0$  representing the sums. Then use Algorithm 2 to return the solution for Opt-threshold. To illustrate the algorithm, suppose that we have items with counts 3, 5, 4, 1. These counts are 011, 101, 100 and 001 in binary. The bitmaps would  $B_2 = 0110$ ,  $B_1 = 1000$  and  $B_0 = 1101$ . At the first pass in the algorithm we have  $A = 0110$ ; there is no change in the second pass, and then  $A = 0100$ —which indicates that only the second item is selected.

For the correctness of the algorithm, note the invariant that, before the  $k^{\text{th}}$  iteration,  $A$  indicates the items whose occurrence counts agree with  $m$  on the most significant  $k - 1$  digits. Clearly this is initially true. If  $\text{AND}(B_{l-k+1}, A)$  is non-empty, the  $k^{\text{th}}$ -most-significant bit of  $m$  must be 1, because  $A$  has at least one item whose occurrence count contains a 1 at the  $k^{\text{th}}$ -most-significant position and because  $m$  is maximal. The assignment statement discards from  $A$  precisely those items whose occurrence counts do not have the 1 in this position. However, if  $\text{AND}(B_{l-k+1}, A)$  is the empty bitmap, then the current items in  $A$  all have counts with a 0 at this position (as must  $m$ ) and the current set of items in  $A$  are not adjusted. At the end of the iteration, in either case, all items in  $A$  have occurrence counts matching  $m$  in their most significant  $k$  digits. After the final iteration, we have an  $A$  that meets the required condition.  $\square$

**Symmetric functions beyond threshold:** We could apply the idea of the SSUM algorithm to general symmetric functions. One could use the sideways sum circuit to compute the  $\lfloor \log 2N \rfloor$ -bit Hamming weights of the inputs followed by a circuit for the corresponding test (e.g., is the result in  $[T_1, T_2]$ ?) in lieu of the  $\geq$  circuit.

In cases where  $N$  is small, we are guaranteed to use few gates. Indeed, Knuth [24, 7.1.2] observes that since he has calculated the minimum number of gates (12) to realize any 5-input Boolean function, we can realize any symmetric Boolean function of  $N \leq 31$  inputs using no more than  $12 + s(N)$  gates, where  $s(N) = 5N - 2\nu(N) - 3\lfloor \log N \rfloor - 3$  is the number of gates that the sideways-sum circuit uses to compute the Hamming weight [24, Prob. 7.1.2.30]. (For instance, if

$N = 31$  we use  $5 \times 31 - 2 \times 5 - 3 \times 4 - 3 = 130$  gates to compute the sideways sum; with at most another 12 we can realize *any* symmetric function. Contrast the 12 to the at-most  $2 \times 5 - 3 = 7$  additional gates that our  $\geq$  circuit uses to compute the threshold function.)

**6.3.2. Compiling or interpreting circuits.** Implementation of the circuit-based approach can be done in several ways. Our companion report discusses three ways. The approach used in our experiments compiles the required circuit into a straight-line program in a byte code having instructions for AND, OR, XOR, RECLAIM, and so forth. (The RECLAIM instruction is inserted by a simple data-flow analysis that determines the last use of an intermediate bitmap; it serves to free memory that would otherwise be occupied by temporaries. Without it, we run out of memory on our largest queries.) A trivial byte-code interpreter then processes this straight-line byte code.

For instance, the SSUM circuit for  $N = 3$  and  $T = 2$  is the carry-out from a full adder:  $(B_1 \oplus B_2) \wedge B_3 \vee B_1 \wedge B_2$ ; the byte-code sequence corresponds to  $t_1 \leftarrow \text{XOR}(B_1, B_2)$ ;  $t_2 \leftarrow \text{AND}(t_1, B_3)$ ; RECLAIM  $B_3$ ; RECLAIM  $t_1$ ;  $t_3 \leftarrow \text{AND}(B_1, B_2)$ ; RECLAIM  $B_1$ ; RECLAIM  $B_2$ ; return  $\text{OR}(t_2, t_3)$ .

For the space analysis of this approach, it appears we calculate and store  $\Theta(N)$  bitmaps. However, we reclaim bitmaps when they are no longer needed, and in practice we may rarely use many more than the initial  $N$  bitmaps provided as inputs. For instance, one of the circuits for  $N = 5$  computed 12 bitmaps in addition to the 5 input bitmaps. However, at no point in the computation were we storing more than 8 bitmaps, or more than 4 temporary bitmaps beyond the initial 5. Since we work with compressed bitmaps, the storage required is data dependent and difficult to determine analytically. The size *without compression or space reclamation* would be  $\Theta(Nr/W)$  words, but  $O(Nr/W)$  is a pessimistic bound when our compiled approach is used with EWAH.

Our companion report describes how one can pre-compile a modest size selection of sideways-sum circuits; the required circuit for a query does not need to be computed on the fly. Thus, in our experiments we time only the execution of the byte code.

#### 6.4. LOOPED algorithm

Given  $N$  bitmaps  $B_1, B_2, \dots, B_N$ , the LOOPED algorithm (see Algorithm 3) seeks to compute the threshold problem for all thresholds  $1, 2, \dots, T$  using corresponding temporary bitmaps  $C_1, C_2, \dots, C_T$ . Let us consider a concrete example:  $B_1 = 0011$ ,  $B_2 = 1110$  and  $B_3 = 1000$  with  $T = 2$ . At first, we process bitmap  $B_1$  and get  $C_1 = 0011$ ,  $C_2 = 0000$ . We then process bitmap  $B_2$  and get  $C_1 = 1111$ ,  $C_2 = 0010$ . We then process the last bitmap to get  $C_1 = 1111$ ,  $C_2 = 1010$ . As with SSUM, the LOOPED approach also combines basic bitmap operations to synthesize the threshold operation.

Our algorithm uses dynamic programming and is based on the following recurrence formula:  $\vartheta(T, \{b_1, b_2, \dots, b_N\}) = \vartheta(T, \{b_1, \dots, b_{N-1}\}) + \vartheta(T-1, \{b_1, \dots, b_{N-1}\}) \cdot b_N$ . I.e., we can achieve a given threshold  $T$  over  $N$  bits, either by achieving it over  $N-1$  bits, or by having a 1-bit for  $b_N$  and achieving threshold  $T-1$  over the remaining  $N-1$  bits. We can use bit-level parallelism to express this as a computation over bit vectors; loops can compute the result specified by the recurrence. Note that  $\Theta(NT)$  bit-vector operations are used, but we need only  $\Theta(T)$  working bitmaps during the computation, in addition to our  $N$  inputs.

The number of binary bitmap operations is  $2NT - N - T^2 + T - 1$  and depends linearly on  $T$ , which is unusual compared with our other algorithms. Referring to Table V, even for  $(N = 4, T = 3)$  this circuit uses twice as many bitmap operations as SSUM. Nevertheless, experiments show that this algorithm can sometime outperform SSUM (see Table VIII). Indeed, the number of bitmap operations is not necessarily a good predictor of performance when using compressed bitmaps. It depends on the dataset.

An Opt-threshold algorithm is easily obtained from LOOPED: first do the calculation with the maximum permitted value of  $T$ —i.e.,  $N$  or  $N-1$ . Then find the maximum value  $i$  such that  $C_i$  is not empty. This algorithm does  $\Theta(N^2)$  bitmap operations, requiring  $\Theta(N^2r/W)$  time if we assume bitmap compression is ineffective.



**Algorithm 3** LOOPED algorithm.**Require:**  $N$  bitmaps  $B_1, B_2, \dots, B_N$ , a threshold parameter  $T$ 

- 1: create  $T$  bitmaps  $C_1, C_2, \dots, C_T$  initialized with false bits
- 2:  $C_1 \leftarrow B_1$
- 3: **for**  $i \leftarrow 2$  **to**  $N$  **do**
- 4:   **for**  $j \leftarrow \min(T, i)$  **down to** 2 **do**
- 5:      $C_j \leftarrow C_j \vee (C_{j-1} \wedge B_i)$
- 6:    $C_1 \leftarrow C_1 \vee B_i$
- 7: **return**  $C_T$

	$B_1$	$B_2$	Count
→	0	0	0
→	0	1	1
→	1	1	2
	1	1	2
	1	1	2
→	0	0	0
→	0	1	1
→	0	1	1

Figure 5. Runs, showing positions where new runs begin (and where the current Hamming-weight count needs to be adjusted).

### 6.5. Exploiting run-length coding: RBMRG

Algorithm RUNNINGBITMAPMERGE (henceforth RBMRG) is a refinement of an algorithm presented in Lemire et al. [11]. See Algorithm 4 and Fig. 5.

The approach considers runs as integer intervals. Each bitmap provides a sorted sequence of intervals. For example, the bitmap  $B_1 = 00111000$  might be viewed as the sequence (bit: 0, range [0, 1]; bit 1, range [2, 4]; bit 0, range [5, 7]). It works for bitmaps that have been run-length encoded.

Heap  $H$  enables us to quickly find, in sorted order, those points where intervals begin (and the bitmaps involved). At such points, we calculate the function on its revised inputs; in the case of symmetric functions such as threshold, this can be quick. As we sweep through the data, we update the current count. Whenever a new interval of 1s begins, the count increases; whenever a new interval of 0s begins, the count decreases. Assuming  $\log N \leq W$ , the new value of a threshold function can be determined in  $\Theta(1)$  time whenever an interval changes. (The approach can be used with Boolean functions in general, but the complexity analysis might differ.)

Every run passes through a  $N$ -element heap, giving a running time of  $O(\text{RUNCOUNT} \log N)$ . One can implement the  $N$  required iterators in  $O(1)$  space each, leaving a memory bound of  $O(N)$ .

The EWAH implementation of RBMRG processes runs of clean words as described. If the interval from  $a'$  to  $a$  corresponds to  $N_{\text{clean}}$  bitmaps with clean runs, of which  $k$  are clean runs of 1s, the implementation distinguishes three cases:

1.  $T - k \leq 0$ : the output is 1, and there is no need to examine the  $N - N_{\text{clean}}$  bitmaps that contain dirty words. This pruning will help cases when  $T$  is small.
2.  $T - k > N - N_{\text{clean}}$ : the output is 0, and there is no need to examine the dirty words. This pruning will help cases when  $T$  is large.
3.  $1 \leq T - k \leq N - N_{\text{clean}}$ : the output will depend on the dirty words. We can do a  $(T - k)$ -threshold over the  $N - N_{\text{clean}}$  bitmaps containing dirty words.

We process the  $N - N_{\text{clean}}$  dirty words as follows. If  $T - k = 1$  (resp.  $T - k = N - N_{\text{clean}}$ ), we compute the bitwise OR (resp. AND) between the dirty words. Otherwise, if  $T - k \geq 128$ , we always use SCANCOUNT using 64 counters (see § 6.1). Otherwise, we compute the number of 1s  $\beta$  in the dirty words. This can be done efficiently in Java since the `Long.bitCount` function on desktop processors is typically compiled to fast machine code.

**Algorithm 4** Algorithm RBMRG.

---

**Require:**  $N$  bitmaps  $B_1, \dots, B_N$  over  $r$  bits, some Boolean function  $\gamma$  such as  $\vartheta(T, \{\cdot\})$

$I_i \leftarrow$  iterator over the runs of identical bits of  $B_i$

$\Gamma \leftarrow$  a new buffer to store the aggregate of  $B_1, \dots, B_N$  (initially empty)

$\gamma \leftarrow$  the bit value determined by  $\gamma(I_i, \dots, I_N)$

$H \leftarrow$  a new  $N$ -element min-heap storing ending values of the runs along with their iterators

$a' \leftarrow 0$

**while** true **do**

  let  $a$  be the minimum of all ending values for the runs of  $I_1, \dots, I_N$ , determined from  $H$

  append run  $[a', a]$  to  $\Gamma$  with value  $\gamma$

$a' \leftarrow a + 1$

**for** iterator  $I_i$  with a run ending at  $a$  (selected from  $H$  as root element) **do**

    increment  $I_i$ ; if  $I_i$  has reached the end, terminate the algorithm

    Update  $\gamma$  with the new value of  $I_i$

    Update the heap  $H$  with the new value of  $I_i$

---

If  $2\beta \geq (N - N_{\text{clean}})(T - k)$ , we use the LOOPED algorithm (§ 6.4), otherwise we use SCANCOUNT again. We arrived at this particular approach by trial and error: we find that it gives reasonable performance.

The algorithm would be a suitable addition to compressed bitmap index libraries that are RLE-based; as a result of this work, we have added it to JavaEWAH [22].

As an extreme example where this approach would excel, consider a case where each bitmap is either entirely 1s or entirely 0s. Then  $\text{RUNCOUNT} = N$ , and in  $O(N \log N)$  time we can compute the output, regardless of  $r$  or  $B$ .

It is easy to obtain an Opt-threshold algorithm based on RBMRG: in a first pass, record the maximum count value seen. This becomes  $T$ , and the usual RBMRG algorithm then answers the query.

## 7. DETAILED EXPERIMENTS

We conducted extensive experiments on the various threshold algorithms, using EWAH compressed bitmaps generated from real datasets. The various bitmaps in our study, even within a particular dataset, vary drastically in characteristics such as density. We discuss this in more detail before giving the experimental results.

### 7.1. Platforms

Experimental results were gathered on a two identical desktops with Intel Core i7 2600 (3.4 GHz, 8 MB of L3 CPU cache) processors with 16 GB of memory (DDR3-1333 RAM with dual channel). Because all algorithms are benchmarked after the data has been loaded in memory, disk performance is irrelevant.

One system was running Ubuntu 12.04LTS with Linux kernel 3.2, and the other ran Ubuntu 12.10 with Linux kernel 3.5. During experiments, we disabled dynamic overclocking (Turbo Boost) and dynamic frequency scaling (SpeedStep). Software was written in Java (version 1.7), compiled and run using OpenJDK (IcedTea 2.3.10) and the OpenJDK 64-bit server JVM. We believe that results obtained on either system are comparable; even if there is some speed difference, the result should not favour one algorithm over another. We used the JavaEWAH software library [22], version 0.8.1, for our EWAH compressed bitmaps. It includes support for the RBMRG algorithm. Our measured times were in wall-clock milliseconds. All our software is single-threaded.

Table VI. Characteristics of real datasets. Overall bitmap density is the number of 1s, divided by the product of the number of rows and the number of bitmaps ( $B/(Nr)$ ).

Dataset	$r$	Attributes	Bitmaps	Average Density	
				Overall	In workload
IMDB-3gr	1 783 816		50 663	$4.1 \times 10^{-4}$	$1.8 \times 10^{-2}$
PGDVD	2 439 448		11 118	$2.9 \times 10^{-4}$	$3.9 \times 10^{-3}$
PGDVD-2gr	3 513 575		755	$2.8 \times 10^{-1}$	$5.4 \times 10^{-1}$
CensusIncome	199 523	42	103 419	$4.1 \times 10^{-4}$	$1.8 \times 10^{-1}$
TWEED	11 245	53	1167	$4.5 \times 10^{-2}$	$2.6 \times 10^{-1}$
Weather	1 015 367	19	18 647	$1.0 \times 10^{-3}$	$8.0 \times 10^{-2}$

## 7.2. Data

Real data tests were done with datasets IMDB-3gr, PGDVD, PGDVD-2gr, CensusIncome, TWEED and Weather. Our first three datasets (IMDB-3gr, PGDVD and PGDVD-2gr) are similar to datasets used in related work [8]. They are *not* indexed as if they were database tables. The last three datasets (Weather, TWEED and CensusIncome) are more representative of content from relational databases and they are indexed as such (see Fig. 2). Performance on some synthetic datasets, and on two other real datasets that were similar to those used here, is discussed in the companion report [7]. Some statistics for each dataset are given in Table VI.

IMDB-3gr is based on descriptions of a dataset used in the work of Li et al. [8], in an application looking for actor names that are at a small edit distance from a (possibly misspelt) name. Our companion report describes how this dataset can be prepared. Each bitmap corresponds to a 3-gram found in some actor’s name. The  $k^{\text{th}}$  bit in the bitmap indicates whether the  $k^{\text{th}}$  actor’s name contains this 3-gram.

The PGDVD dataset has a bitmap for each of 11 118 files on the Project Gutenberg DVD [34]. Each bitmap represents the vocabulary set found in that file (the total vocabulary had over 2.4 million words). Details and a potential application of this dataset are described in the companion report.

PGDVD-2gr is similar to IMDB-3gr except that, instead of actor names, we formed 2-grams from chunks of text from the Project Gutenberg DVD. Each chunk was obtained by concatenating paragraphs until we accumulated at least 1000 characters. We rejected any paragraph with over 20 000 characters—this protected us from some non-text content (e.g., the digits of  $\pi$ ) on the Project Gutenberg DVD.

We also chose three more conventional datasets in the context of relational databases. Two have many attributes, CensusIncome [11, 35] and TWEED [36]. The former is a census extract; the latter is a small dataset containing historical information on terrorist attacks in Europe, for which we used all attributes, rather than the projection used by Webb et al. [37]. We also used the entries for September 1985 of a larger dataset (Weather) [38, 39]. This particular month has been used previously [39], although the previous use had projected only 9 attributes, whereas we used all of them. We selected just one month of data because the full dataset (123 million rows) caused several of the tested algorithms to run out of memory. It would have been difficult to report meaningful aggregate results with such failures.

A bitmap index was built for each conventional dataset, and it had a bitmap for every attribute value. Row reordering can improve RLE-compressed bitmap indexes [11], but it is not always possible. Our indexes used the given (unsorted) row order. In CensusIncome, one attribute is responsible for 99 800 of the bitmaps; the remaining 3619 bitmaps are much denser than these 99 800. Together, our real datasets cover a range of application areas, lengths, widths and densities.

By design, our work does not consider external-memory indexes on very large datasets. Thus our datasets are chosen to fit in RAM. However, even if our machines had more than 16 GB of RAM, we might still want to partition the problems so that bitmaps do not span much more than a few million bits, to alleviate caching issues.

### 7.3. Queries Used

To assess our algorithms, we generated a random workload of 10 000 Many-Criteria and Similarity queries (see § 4).

- To generate a Many-Criteria query, we randomly chose a dataset. Many-Criteria queries do not make much sense for IMDB-3gr, PGDVD or PGDVD-2gr. For instance, almost all 3-grams have extremely sparse bitmaps and empty results can be expected, even with  $N$  large and  $T$  small. Therefore, we chose the dataset with equal probability from {CensusIncome, TWEED, Weather}. Having determined the dataset, we chose  $N$  next. We used a discretized log-uniform distribution with  $\log N \sim U[\log 3, \log 1000]$ , which resulted in a workload where small values of  $N$  were more common, but large values of  $N$  sometimes occurred. We feel this was more realistic than having generated values from 3 to 1000 uniformly. We then chose (uniformly at random, with replacement)  $N$  attributes on which criteria were established, by choosing one of their bitmaps uniformly. We finally randomly chose an integer threshold  $T$  uniformly from  $[2, N' - 1]$ , where  $N'$  is the number of attributes on which criteria had been established.
- We considered Similarity queries with  $n$  prototypes (henceforth  $\text{Similarity}(n)$ ). For such a query, we selected (with equal probability) one of our datasets. Then we chose  $n$  distinct prototypes  $\{r_i \mid i \leq n\}$ , each represented by a row identifier chosen uniformly from  $[0, r)$ . We then found the set of bitmaps matching at least one of them,  $\bigcup \{B_i \mid \exists j \text{ such that } B_i[r_j] = 1\}$ ; we have that  $N$  was the number of matching bitmaps.

The probability of choosing a Many-Criteria query was 50 %, whereas the probabilities of  $\text{Similarity}(1)$ ,  $\text{Similarity}(5)$ ,  $\text{Similarity}(10)$ ,  $\text{Similarity}(15)$  and  $\text{Similarity}(20)$  queries were 10 % each.

We agree with Jia et al. [14] that it does not make sense to time queries whose answers are empty. Regardless whether we had a Many-Criteria or a Similarity query, if the answer to the threshold query was empty and  $T > 2$ , we chose (uniformly at random) a new value of  $T$  between 2 and the existing value of  $T$ . If the threshold query had an empty answer when  $T = 2$ , we discarded the query and generated a new one.

Of our 10 000 queries, there were 63 queries with  $N > 1000$ : the maximum value of  $N$  was 11 116 whereas the average  $N$  was 190. The maximum value of  $T$  was 8405, but the average was 45. The largest set of input data, in terms of storage, was 185 MB; in terms of cardinality it was 740 million items.

The bitmaps involved in our queries are denser than the average bitmap. Indeed, the last two columns in Table VI differ: the first shows the average density of bitmaps from the dataset, whereas the second shows the average density of the bitmaps actually selected in our workload. We see that the latter are denser (anywhere from twice as dense to 1000 times denser). This is a consequence of how we pick the queries.

- Many-Criteria queries tend to choose dense bitmaps because the sparsest bitmaps frequently come from the same (high cardinality) attribute, and all attributes are given an equal probability.
- For Similarity queries, denser bitmaps are more likely to appear in  $\bigcup \{B_i \mid \exists j \text{ such that } B_i[r_j] = 1\}$ .

**Choosing  $\mu$  for DSK:** The DSK algorithm requires a tuning parameter  $\mu$ , which depends on the dataset. Li et al. [8] sketch a process for choosing  $\mu$ :

- For each dataset, select a representative workload of queries.
- For each query, execute DSK with various choices of  $\mu$ , recording the  $\mu$  that produced the fastest answer for that query.
- Average the recorded  $\mu$  values for a dataset.

We followed their approach. For the workload, we generated 1000 queries using the random query generation process already described. We tried up to 20 values of  $\mu$  for each query, using

the relationship  $L = T/(\mu \log M + 1)$  given by Li et al. ( $M$  is the cardinality of the largest bitmap) to choose  $\mu$  values. When  $T \leq 20$ , we tried  $L = 1, L = 2, \dots, L = T - 1$ . Otherwise, we tried all values in  $[T - 5, T) \cup \{\lceil \frac{T-6}{15} i \rceil \mid i \in [1, 15]\}$ . For IMDB-3gr, PGDVD, PGDVD-2gr, CensusIncome, TWEED and Weather, the respective  $\mu$  values were 0.164, 0.110, 0.00416, 0.0321, 0.0350 and 0.0587.

**Competitions:** We assess the effectiveness of the various algorithms by measuring their wall-clock times on the queries in our workload. Each query can be viewed as a competition between algorithms.

Unfortunately, for some of the larger queries, w2CTI (§ 6.1.1) was not able to complete without running out of memory. It is unfair to give the algorithm a nearly infinite running time when trying to compute its aggregate performance over the workload. However, it is also unfair to omit the running time from an average, as it is excusing a result where even a good algorithm would take a long time. Our solution is to assign the running time of the slowest algorithm that *did* complete the competition.

#### 7.4. Scalability of Algorithms

As previewed in Table III, the various algorithms' running times are all affected<sup>||</sup> by  $N$ . Some are affected by  $T$  and others are highly sensitive to the characteristics of the datasets being processed. The companion report describes in detail these effects for each algorithm, but a few examples here illustrate these effects.

For chosen values of  $N$ , we took 100 queries on CensusIncome and, for each algorithm, averaged their running times. These are majority queries: threshold queries with  $T = \lceil N/2 \rceil$  and  $N$  odd. (Unlike our normal workload, we have a mixture of queries returning empty and non-empty results.) In this scenario, we have that  $r$  and  $W$  are fixed while  $B$ ,  $B'$ ,  $T$  and  $N - T$  grow with  $N$ . From Table III, we might expect (using an admittedly naïve analysis) the running time of SCANCOUNT and SSUM to grow linearly ( $N$ ), the running time of MGOPT, DSK and RBMRG to grow as  $N \log N$  and the running time of w2CTI and LOOPED to grow quadratically ( $N^2$ ). Table VII shows how several algorithms behaved in this particular test as  $N$  was changed. The column for  $N = 3$  reports a measured time, whereas every other column shows the time increase from the column to its left—i.e., as its input is increased threefold. SCANCOUNT gains an advantage with larger  $N$ , because the cost of initializing and reading the counters is independent of  $N$ . On the other hand, SSUM starts out faster than SCANCOUNT but scales poorly at first ( $\approx 8$ ) though it scales as well as SCANCOUNT for large values of  $N$  ( $\approx 3$ ). But for smaller values of  $N$ , SCANCOUNT has a lower growth rate. Somewhere between  $N = 27$  and  $N = 81$ , SCANCOUNT becomes faster than SSUM. For LOOPED, when  $N$  triples, the running time starts by increasing roughly 9-fold, as one would expect from an algorithm that does  $\Theta(NT)$  bitmap operations. The growth drops off as  $N$  increases (to  $\approx 5$ ), indicating that the individual bitmap operations have become less costly. Still, LOOPED's growth rate exceeds the others'. Comparing DSK to MGOPT, on this dataset there is little difference for small  $N$ , but as  $N$  grows we see DSK grow more slowly. If we compare RBMRG and w2CTI, the latter scales better (growth factor less than  $3\times$ ). However, the difference is not enough to overcome the order of magnitude speed advantage that RBMRG starts with.

Fig. 6 shows the effect of varying  $T$ , on one particular set of 171 bitmaps corresponding to a Similarity(1) query. Absolute times are shown, but on a logarithmic scale. For this collection of bitmaps, we got best results from LOOPED at  $T = 2$ , then RBMRG from  $T = 3$  to  $T \approx 70$ , then SSUM (whose time is about 500ms) until  $T = 165$ , after which DSK was fastest. The potentially enhanced pruning of DSK over MGOPT was not manifest until  $T = 145$  on this dataset, whereas for Table VII even majority queries showed an advantage for DSK.

<sup>||</sup>For table entries (such as that for SCANCOUNT) where  $B$  is given but  $N$  is not explicit, note that  $B$  grows as  $N$  grows: given a set of  $N$  bitmaps with  $B$  1s, if a new non-empty bitmap is added, the total number of 1s increases.

Table VII. Effect of  $N$  on running times of the algorithms. Many-Criteria majority queries were used (i.e.,  $T = \lceil N/2 \rceil$ ) and the chosen dataset is CensusIncome.

Algorithm	Time(ms)	Time( $N$ )/Time( $N/3$ )				
	$N = 3$	$N = 9$	$N = 27$	$N = 81$	$N = 243$	$N = 729$
RBMrg	$3.41 \times 10^{-2}$	3.1	3.3	5.0	3.6	3.5
ScnCnt	$2.16 \times 10^{-1}$	1.6	2.6	2.9	2.7	2.8
SSum	$2.28 \times 10^{-1}$	1.2	2.6	2.7	3.2	3.1
Looped	$3.23 \times 10^{-2}$	9.5	6.8	5.9	5.2	5.0
DSk	$1.96 \times 10^{-1}$	1.4	3.7	4.4	4.5	3.0
w2Ctl	$2.46 \times 10^{-2}$	8.3	3.9	3.4	3.0	2.8
MgOpt	$3.48 \times 10^{-1}$	1.9	2.7	3.0	2.9	2.8

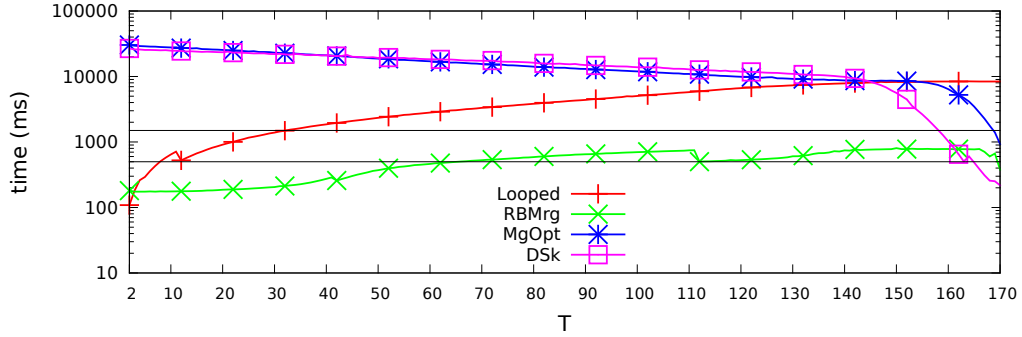


Figure 6. Effect of  $T$  on the running times of several algorithms.  $N = 171$ , and the dataset is PGDVD-2gr. Other algorithms were less affected by  $T$ . The SCANCOUNT algorithm took about 1500 ms for all values of  $T$  and SSUM always took about 500 ms. The w2Ctl algorithm dropped steadily from about 13 000 ms for small  $T$  to about 10 000 ms for the largest  $T$  values.

### 7.5. Comparing Algorithms

Because the state of our system varies slightly over time, we make an error when measuring the time required by the implementation of an algorithm. We think of the *true* performance of the implementation of the algorithm as its best possible speed on a given query. We can measure this best possible speed with little error by repeating the execution hundreds of times. However, given our 10 000 queries and 7 algorithms, these repeated tests would require more than a year to complete. Thus we tested each algorithm on each query only a few times.

Moreover, when comparing algorithms, we did not merely want to decide whether an algorithm is superior to another; for this purpose a standard statistical test would have sufficed. Instead we wanted to compare the results numerically, so we first estimated our measurement error by generating 5 random queries. Because timings errors are always additive, given a query, we ran each algorithm on each query 200 times. The minimum timing is assumed to be error-free: the fastest test out of 200 tests is a good approximation of the fastest possible result. Any larger timing is in error. We have 6 datasets and 7 algorithms, so we collected  $200 \times 6 \times 7$  measurement errors per query. We found that the 99<sup>th</sup>-percentile error was less than 10 % for the 5 reference queries.

We then considered our set of 10 000 queries. For each query, dataset and algorithm, we fixed a number of repetitions so that the total running time is at least 1 s. Supposing the measured running times on some query are  $t_1$  and  $t_2$  for two given algorithms, we say that the first algorithm is faster only if  $t_1 < 0.8 \times t_2$ . We anticipate mis-identifying a superior algorithm less than 1 % of the time.

Table VIII compares each pair of algorithms. The cell associated with the row for algorithm  $\mathcal{A}_1$  and the column for algorithm  $\mathcal{A}_2$  gives the number of times that  $\mathcal{A}_1$  had performance superior to that of  $\mathcal{A}_2$ .

For all cases when  $t_1 \leq t_2$ , we record the percentage improvement measured. (A percentage improvement of  $x$  means that  $\mathcal{A}_1$  is  $1/(1-x)$  times faster than  $\mathcal{A}_2$ . That is, improvements of

Table VIII. Percentage of competitions where the row’s algorithm was at least 20 % faster than the column’s algorithm, and beneath it, the percentage improvements from the row’s algorithm. We show the median, 75<sup>th</sup>-percentile, and maximum percentage improvement. An improvement of 99 % means at least 100× speed.

The final column shows the percentage of cases where the row’s algorithm was measured to be fastest.

vs	RBMRG	SCNCNT	SSUM	LOOPED	DSK	w2CTI	MGOPT	fastest
RBMRG		77 % 64 79 99	88 % 56 67 96	93 % 82 89 99	96 % 89 96 99	98 % 89 95 99	99 % 88 94 99	70 %
SCNCNT	13 % 37 61 79		34 % 30 55 91	66 % 75 88 99	84 % 84 90 96	93 % 80 84 98	86 % 82 86 98	18 %
SSUM	3 % 17 22 95	38 % 41 59 99		72 % 65 79 99	82 % 83 92 99	88 % 80 87 99	88 % 78 88 99	6 %
LOOPED	3 % 34 50 75	26 % 58 77 99	13 % 37 64 91		61 % 73 90 99	60 % 66 86 99	55 % 65 87 99	4 %
DSK	1 % 20 42 67	11 % 53 74 98	9 % 33 55 83	27 % 57 77 99		21 % 30 62 99	14 % 19 39 96	1 %
w2CTI	1 % 19 28 74	5 % 51 71 97	6 % 34 53 89	26 % 51 71 99	54 % 34 48 98		33 % 21 35 98	1 %
MGOPT	0 % 10 18 39	10 % 55 76 98	5 % 34 52 82	30 % 47 65 99	40 % 21 31 74	22 % 25 51 99		0 %
fastest	0 11 95	58 77 99	58 71 96	83 90 99	90 96 99	90 95 99	89 94 99	

99 %, 90 %, 80 %, 50 % indicate that we have 100×, 10×, 5× and 2× the speed.) To assess these performance improvements (ignoring the possibility of measurement error), we show the time reductions that could be obtained for the query, by using  $\mathcal{A}_1$  instead of  $\mathcal{A}_2$ . We show the 50<sup>th</sup>- and 75<sup>th</sup>-percentile and maximum time reductions in percentage.

We round percentage reductions down, thus percentage reductions of 99 mean speedups of *at least* 2 orders of magnitude are possible by switching algorithms. Note that even the weakest algorithm outperforms each of the others (excepting RBMRG), even if rarely. The final column in the table shows the number of workload queries where the row’s algorithm was the best (ignoring possible measurement error). The superior algorithms are RBMRG (70 % of the queries), SCANCOUNT (18 %) and SSUM (6 %). (Also, LOOPED is best on 4 % of the queries, typically when  $T$  is small.)

The final row represents the case where an oracle picks the fastest algorithm for each query. As expected, because RBMRG is best 70 % of the time, the median of the percentage improvements is zero for this algorithm. The final row shows that *every* algorithm performs badly on at least one instance (e.g., RBMRG is beaten by 95 % once which means that another algorithm is 20× faster). We see that SCANCOUNT, DSK, LOOPED, MGOPT and w2CTI are sometimes at least two orders of magnitude slower than necessary. Algorithms SSUM and RBMRG are less disastrous, being approximately 20–25 times slower in the worst cases. At the 75<sup>th</sup> percentile level RBMRG is the clear winner (which is expected given that it is best 70 % of the time). SCANCOUNT and SSUM are similar : each is typically about four times slower than the best algorithm. Although SCANCOUNT is the fastest algorithm more than three times as often as SSUM, note that SSUM and SCANCOUNT are roughly tied with one another (each winning about 5000 times) and the percentile improvements favor SSUM over SCANCOUNT.

**Beating SCANCOUNT:** In § 5 we suggested that SCANCOUNT could be beaten; indeed, we can see this by inspecting the SCANCOUNT column in Table VIII. To be more precise, our workload contained a query that, compared to SCANCOUNT, was answered 540× faster using RBMRG, a query that was 570× faster with SSUM, one that was 740× faster with LOOPED, one that was 72× faster using DSK, one that was 36× faster with w2CTI, and one where MGOPT was 79× faster. SCANCOUNT can be beaten by orders of magnitude.

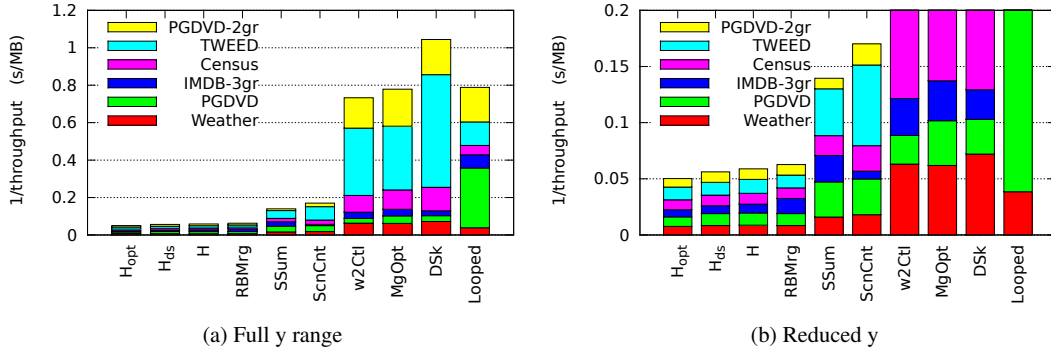


Figure 7. Aggregate throughput on each dataset. Bar height represents the number of seconds for a workload containing 1 MB of bitmap data from each dataset.  $H_{opt}$ ,  $H_{ds}$  and  $H$  are discussed in § 8.2

### 7.6. Performance Across Workload Subsets

Table IX shows the total time taken by each algorithm across the entire workload, or across a portion of the workload meeting certain criteria shown in the first column. Table IX shows the effect of large  $N$ , small  $T$  or  $N$ , the kind of query, or the dataset. The companion report contains additional results on the scaling of each algorithm.

The table shows that LOOPED, DSK, MGOPT and w2CTI can have some extremely expensive queries, although fewer than 25 % of the queries are extremely expensive. The apparent preference for RBMRG toward the top of the table partly breaks down when we examine individual datasets at the bottom of the table. The large size of PGDVD-2gr and the excellent performance of SSUM and RBMRG on this large dataset act together to dominate the overall results. Also, results are dominated by larger values of  $N$ , despite our attempt to generate workloads so that small- $N$  queries were more frequent than large- $N$  ones. As well, costs are dominated by Similarity queries; while equal in number to Many-Criteria queries in our workload, they include the queries with the largest values of  $N$ .

To visualize or aggregate this data, we should consider that the workload involves datasets of widely different size: there is three orders of magnitude between the total volume of data for our TWEED queries and our PGDVD-2gr queries. Instead of merely timing the queries, we measure their throughput: amount of input data divided by the time necessary to complete the query, expressing the result in MB/s. Given an algorithm and a dataset, we use the harmonic mean to obtain an aggregate throughput value. However, for display purposes it is convenient to show the reciprocal throughput. For instance, the stacked bar charts in Fig. 7 can be viewed as representing times (in seconds) on some hypothetical workload in which 1 MB of bitmap data had been processed by the queries for each dataset. For our workload, RBMRG, SCANCOUNT and SSUM are strongly preferred to the others. Figure 8 shows that, for our relational datasets—the only ones that were used with both Many-Criteria and Similarity queries—RBMRG is the clear winner.

In several applications we can expect that  $N$  will not be particularly large, or that typical queries will usually have  $T \approx N$ . Figure 9 shows the results for these cases. Even though PGDVD-2gr—a dataset that seemed to favour SSUM on the full workload—had no queries with  $N \leq 16$ , SSUM outperformed SCANCOUNT. On a typical query with  $N$  small, the cost of initializing and scanning the  $r$  counters is being amortized over a small volume of bitmap data. We also see that LOOPED is a viable algorithm for  $N \leq 16$ .

Figure 9 also shows the situation for the workload queries where  $T \geq 0.75N$ . This situation is one where pruning-based algorithms such as w2CTI, MGOPT and DSK can excel. Indeed, we see them doing well on IMDB-3gr, Weather and (for w2CTI and DSK) PGDVD. However, there are other datasets where they still perform badly. Altogether, on workloads similar to ours, the benefits from pruning do not seem to be worth the risks of using these algorithms. Nevertheless, there are some applications where the requirement for a large  $T$  can be met. For instance, using the



Table IX. Total time to process queries of various groups. The top line of each group is the total time. Then four lines give the 25<sup>th</sup>-, 50<sup>th</sup>-, 75<sup>th</sup>-percentile, and maximum query times. Values for RBMRG are absolute (seconds for total time; ms for percentile values). Values for all other algorithms are relative—the measured time has been normalized by dividing it by the corresponding time for RBMRG.

data	RBMRg	ScnCnt	SSum	Looped	DSk	w2CtI	MgOpt
$N \leq 15$	$3.1 \times 10^0$	3.42	1.98	2.62	5.43	6.11	4.75
	0	11.9	4.9	3.8	18.8	24.7	15.0
	1	5.0	1.6	1.9	3.4	4.7	3.1
	2	4.3	1.9	2.4	4.5	5.8	4.2
	17	3.7	2.3	4.0	19.0	9.0	15.2
$N \geq 16$	$1.7 \times 10^3$	1.53	1.24	27.43	14.59	12.45	15.13
	3	3.0	2.9	5.8	16.7	9.5	12.2
	25	2.1	2.3	4.5	11.3	9.9	11.4
	140	0.8	1.7	4.6	4.9	4.5	4.9
	2609	1.2	2.3	801.1	25.9	29.0	22.6
$T < 5$	$4.0 \times 10^1$	1.97	2.13	1.24	21.17	11.77	19.70
	0	4.9	2.0	1.5	7.2	8.1	5.3
	1	4.8	1.9	1.8	6.8	7.4	5.8
	8	2.4	2.2	1.8	10.2	8.0	8.4
	696	3.8	2.2	1.0	92.6	40.8	84.6
Similarity( $n$ )	$1.4 \times 10^3$	1.64	1.21	31.41	15.89	13.88	16.82
	6	2.3	2.6	3.9	8.2	4.1	5.9
	32	1.9	2.2	6.1	13.0	9.4	11.8
	256	0.5	1.7	5.0	2.7	2.7	2.8
	2609	1.2	2.3	801.1	25.9	29.0	22.6
IMDB-3gr	$1.8 \times 10^2$	0.38	1.58	5.67	1.83	2.43	2.53
	84	0.6	1.8	4.1	1.8	2.1	3.3
	218	0.4	1.5	3.4	1.8	2.4	2.7
	320	0.4	1.6	5.8	1.8	2.4	2.4
	597	0.3	1.3	10.6	1.9	2.6	2.7
PGDVD	$7.9 \times 10^1$	0.46	2.60	253.73	2.63	2.71	6.84
	3	3.6	2.7	4.5	2.8	2.5	3.2
	11	1.4	3.6	8.7	2.5	2.1	3.2
	42	0.7	3.5	13.6	2.3	1.9	3.5
	2609	0.3	2.3	801.1	4.3	11.5	13.5
PGDVD-2gr	$1.1 \times 10^3$	1.91	1.00	20.91	19.33	16.81	20.23
	807	3.1	1.4	9.6	14.9	19.8	23.1
	1504	1.7	0.9	12.6	15.9	14.9	15.1
	1763	1.5	0.9	26.2	20.7	14.4	19.3
	2051	1.5	1.0	38.5	32.9	36.9	28.7
CensusIncome	$8.6 \times 10^1$	1.38	1.53	6.41	10.73	6.57	8.84
	3	3.1	2.5	2.6	10.3	10.7	9.3
	15	2.7	2.3	3.4	18.3	12.8	15.7
	33	1.9	1.9	6.2	16.4	9.6	13.1
	291	0.8	1.0	17.7	9.4	3.7	7.7
TWEED	$1.9 \times 10^0$	3.49	3.16	16.18	51.83	22.62	29.92
	0	8.3	6.2	4.0	40.5	39.1	21.3
	0	6.0	3.9	6.4	63.8	35.1	31.2
	1	3.5	3.2	10.7	52.0	23.2	31.4
	7	2.0	2.1	43.0	42.2	13.4	37.0
Weather	$2.4 \times 10^2$	1.07	1.55	5.90	7.91	5.60	6.53
	8	3.2	2.6	3.2	7.6	10.0	7.4
	39	2.2	2.4	3.2	13.1	10.1	11.8
	87	1.4	2.0	5.1	8.5	7.4	7.9
	738	0.8	1.1	16.5	10.8	5.5	9.6
All	$1.7 \times 10^3$	1.53	1.24	27.38	14.58	12.44	15.12
	1	3.7	2.7	4.9	18.6	15.7	13.4
	13	2.3	2.5	4.5	10.9	8.6	9.7
	88	1.2	1.9	5.4	6.9	6.1	6.8
	2609	1.2	2.3	801.1	25.9	29.0	22.6

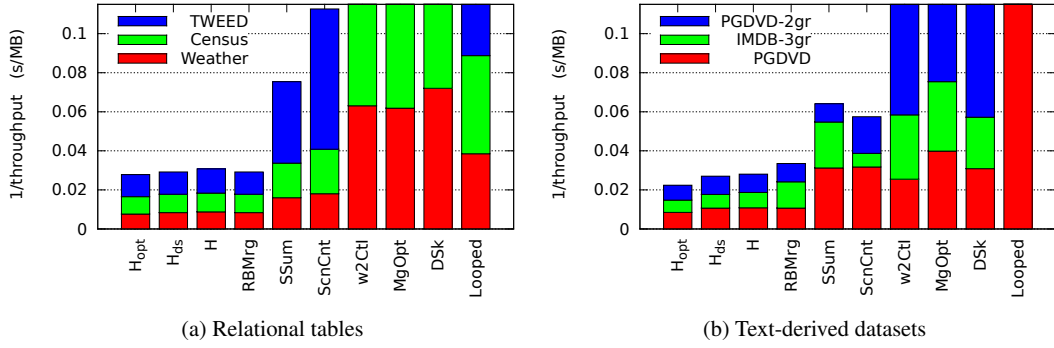


Figure 8. RBMRG excels on relational data. On the text-derived datasets, SCANCOUNT and SSUM were sometimes better.

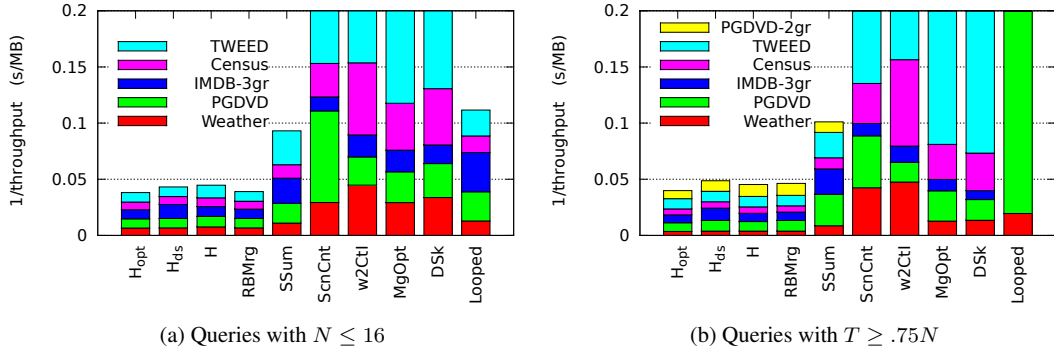


Figure 9. Normalized workload times for queries with small  $N$  and for queries with  $T \geq .75N$ .

formula of Sarawagi and Kirpal [9] with strings of length  $N = 64$ , if we are interested in finding the strings of edit distance at most two from some target using trigrams, the appropriate threshold is  $T = 64 + 3 - 1 - 2 \times 3 = 60$ .

**7.6.1. The advantage of RBMRG.** Our results show that RBMRG was usually the fastest algorithm, especially over datasets coming from relational tables. This speed advantage is due to fewer executed instructions, rather than cache effects: experiments showed that the processor executed about two instructions per cycle (IPC) for all implementations. (We saw 1.9 IPC for SCANCOUNT; 2.0 for w2CTI; 2.1 for SSUM, LOOPED and RBMRG; and 2.2 for MGOP and DSK.)

One reason for the advantage of RBMRG is that it ends up solving a threshold problem over the dirty words, and our implementation adaptively switches between algorithms LOOPED and SCANCOUNT. In essence, it gets a benefit from RLE encoding, and then combines the strengths of two other efficient algorithms. An initial implementation had done a naïve computation (iterating over all bit positions) and this implementation of RBMRG was usually not competitive with SCANCOUNT and SSUM. Solving the threshold subproblem effectively on the dirty words was crucial, and our hybrid of SCANCOUNT and LOOPED made the revised implementation fast.

## 8. HYBRID ALGORITHMS

Our success in handling dirty words adaptively suggests that an adaptive, hybrid approach might also be a better way to solve threshold problems on compressed bitmaps.

Table X. Running time estimates for good algorithms (this excludes w2CTI, MGOPT and DSK).

Algorithm	Time complexity estimate	Fitted coefficients
SCANCOUNT	$c_{sc,1} \times r + c_{sc,2} \times B$	$c_{sc,1} = 2.734 \times 10^{-5} \pm 4.5 \times 10^{-7}$ $c_{sc,2} = 3.464 \times 10^{-6} \pm 3.2 \times 10^{-9}$
LOOPED	$c_{LOOPEd} \times T \times \text{EWAHSIZE}$	$c_{LOOPEd} = 1.499 \times 10^{-6} \pm 3.6 \times 10^{-9}$
SSUM	$c_{SSUM} \times \text{EWAHSIZE}$	$c_{SSUM} = 1.039 \times 10^{-5} \pm 4.2 \times 10^{-8}$
RBMrg	$c_{RBMRG} \times \text{EWAHSIZE} \times \ln N$	$c_{RBMRG} = 1.599 \times 10^{-6} \pm 5.3 \times 10^{-9}$

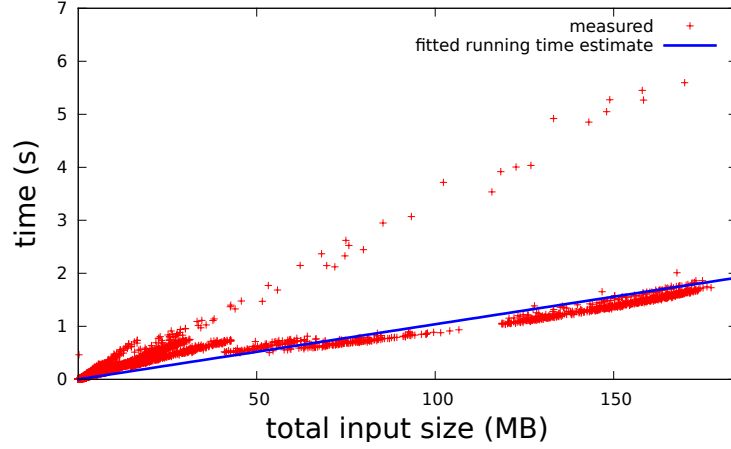


Figure 10. The running time for SSUM depends on the total compressed size of the input bitmaps. We show a least-squares line (passing through the origin) to fit this data.

### 8.1. An Execution-Time Model

To guide an adaptive algorithm, we need to estimate the running times of the more promising algorithms, in terms of the limited data that a DBMS might be expected to maintain.

Table X shows estimates of the running-time functions over our workload. They were derived by least-squares fitting our running-time bounds in § 6 and Table III to the measured times for the competitions in the workload. To account for bitmap compression, we substitute EWAHSIZE where  $Nr/W$  occurred in Table III. Given a bound of  $O(f(x_1, x_2, \dots, x_k))$ , we modeled the running time as  $cf(x_1, x_2, \dots, x_k)$  and fitted  $c$  according to the measured running times. Algorithm SCANCOUNT had two independent terms and we used a separate constant for each term. Also, for RBMRG, we felt it would be unreasonable to expect the RUNCOUNT of the bitmaps to be cataloged. Instead, we used EWAHSIZE as a proxy.

Only SSUM’s function has one independent variable and is thus suited for display; this function is plotted in Fig. 10. It appears that the measured data consists of six different and fairly linear sequences. Presumably there is a different linear sequence of points for each of the datasets, and the slope for each depends on characteristics of the dataset that the running-time function does not consider. Based on Fig. 10, we see that our estimate tends to be optimistic for smaller inputs. On larger inputs, there is one dataset where we will significantly underestimate the running time, but for others, we will be pessimistic.

### 8.2. Algorithms

We experimented with hybrid algorithms H and  $H_{ds}$ , described below. For comparison purposes,  $H_{opt}$  is the hybrid algorithm that always chooses the fastest algorithm for any query.

**Hybrid algorithm, H.** Since we have multiple alternative algorithms for the same problem, there are sophisticated approaches for choosing the best algorithm for a given application [40]. However, we can get reasonably good performance with two simple approaches for choosing the appropriate algorithm.

Our first approach, hybrid algorithm H, evaluates the running-time estimates given in Table X. It then selects the algorithm predicted to run fastest. Mathematically, the estimates for LOOPED, SSUM and RBMRG are similar; if we know to avoid SCANCOUNT, we can algebraically simplify to the following decision procedure that depends only on  $N$  and  $T$ , and not on dataset parameters:

```

if (T<=6) //Looped vs SSum
  if (0.94*T < Math.log(N)) Looped() //Looped vs RBMRg
  else RBMRg()
else // SSum gets to compete with RBMRg
  if (N <= 665) SSum()
  else RBMRg()

```

Intuitively this does not appear to choose RBMRG as often as we might expect: there are many cases where  $N$  is smaller than 665 and where RBMRG is faster than SSUM. When we tried excluding SSUM, it improved H by 13%, and we therefore adopted this policy when we know not to use SCANCOUNT:

```

if (T<=6) AND (0.94*T < Math.log(N))
  Looped()
else RBMRg()

```

A weakness of this approach is that it is based explicitly on the performance of our particular test computers. While slightly inaccurate estimates may not lead to bad decisions, those using this approach on systems that differ significantly should conduct their own benchmarks and adjust the coefficients.

**Adjust-by-dataset algorithm,  $H_{ds}$ .** Faced with a collection of queries over disparate datasets, an obvious approach is to select the algorithm entirely on the basis of the dataset. Perhaps some initial profile runs would be used to select the algorithm to be used consistently on a dataset. This  $H_{ds}$  approach was tested; on our workload we used SCANCOUNT for all queries against IMDB-3gr, SSUM for all queries against PGDVD-2gr, and RBMRG for all other queries. (We chose this combination by inspecting Fig. 7.)

**Optimal hybrid algorithm,  $H_{opt}$ .** For comparison purposes, we can determine the effect of the optimal hybrid algorithm,  $H_{opt}$ , which always selects the best algorithm for any competition as an oracle would. Since we have already run every algorithm during the competition, this is easy for us to do. Of course, our hypothetical engineer will not have this information— $H_{opt}$  exists only to make comparisons.

### 8.3. Evaluation of Hybrid Algorithms

Looking at Figs. 7–9, results for  $H_{opt}$  indicate that choosing the correct alternative for a query gave a result 25% faster than the best overall algorithm (RBMRG) for the workload. (And our engineer would not have known which single algorithm to have chosen.) Also,  $H_{ds}$  did nearly as well. The gap between hybrid algorithms H and  $H_{opt}$  shows that there is a little room for improved decisions (16%). Nevertheless, H was able to outperform (by 13%) RBMRG on the overall workload. On the workload subsets in Fig. 9, its performance was not much worse (13% worse and 1% better) than the single best algorithm that our engineer might have chosen for the workload. Basing H on more sophisticated predictions than those obtained from Table X, it may be possible to approach the performance of  $H_{opt}$  even more closely. (The maximum possible gains in these two cases would be 13% and 31%.)

#### 8.4. Easier Implementation

Whereas LOOPED and SCANCOUNT have simple implementations, algorithm SSUM has a relatively difficult implementation. (However, the companion report describes a lower-performance version, CSVCKT, with a much simpler implementation.) Algorithm RBMRG requires a detailed knowledge of the internal workings of a RLE-compressed bitmap representation and is best added by the maintainers of a compressed bitmap package. Thus, it is reasonable to look at the tradeoffs that come from easier-to-implement hybrid algorithms that omit SSUM, RBMRG, or both. Our comparison corresponds to bar heights in Fig. 7.

Omitting RBMRG increased the time by 97 %. Allowing neither RBMRG nor SSUM increased H's time by 159 %. This shows that SSUM can replace RBMRG, but only partially.

### 9. CONCLUSION AND FUTURE WORK

To our knowledge, bitmap indexes have never been used for computing thresholds (or other symmetric functions). We reviewed several novel and several known algorithms for this problem. We found that a novel algorithm (RBMRG) was generally superior to alternatives, sometimes being orders of magnitude faster.

Although RBMRG could be considered the overall winner, each algorithm examined was weak in some circumstances. However, we combine them in a hybrid algorithm that improves on any individual algorithm. In future work, we might create better hybrid algorithms, perhaps by applying machine-learning processes to choose the fastest threshold algorithm [41, 42].

Our work has considered  $N$  values up to a few thousand (at most 11 116). Yet datasets whose indexes have millions of bitmaps are not out of the question. Would there be applications where a threshold computation with  $N = 1\,000\,000$  would be useful? If so, which algorithms should be used? The circuit-based algorithms will become infeasible for extremely large  $N$ , but which of the others can be used? Can new algorithms be developed for this case?

When possible, data should be indexed in sorted order [11]: this can improve compression and processing speed. Some algorithms might benefit more than others from sorting, and this warrants further investigation.

Finally, algorithms can be parallelized, and while most of our threshold computations take only a few milliseconds, some take tens of seconds. If we try extremely large  $N$  values, this may increase. At some point, it may become important to have one threshold computation run faster than is possible using a single core. For multicore processing, a particular challenge with current architectures is that all cores compete for access to L3 and RAM. E.g., this means that it is best if intermediate results fit in L2 cache. It might be advisable to partition the problems.

### REFERENCES

1. Antoshenkov G. Byte-aligned bitmap compression. *Data Compression Conference, DCC'95*, IEEE Computer Society: Washington, DC, USA, 1995; 476, doi:10.1109/DCC.1995.515586.
2. Culpepper JS, Moffat A. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems* Dec 2010; **29**(1):1:1–1:25, doi:10.1145/1877766.1877767.
3. Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, *et al.*. C-Store: a column-oriented DBMS. *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB'05*, ACM: New York, NY, USA, 2005; 553–564.
4. Thomsen C, Pedersen TB. A survey of open source tools for business intelligence. *Integrations of Data Warehousing, Data Mining and Database Technologies: Innovative Approaches*, Tanian D, Chen L (eds.). chap. 10, IGI Global: Hershey, PA, USA, 2011; 237–257, doi:10.4018/978-1-60960-537-7.ch010.
5. MacNicol R, French B. Sybase IQ Multiplex — designed for analytics. *Proceedings of the 30th International Conference on Very Large Data Bases, VLDB'04*, VLDB Endowment, 2004; 1227–1230, doi:10.1016/B978-012088469-8.50111-X.
6. O'Neil P, Graefe G. Multi-table joins through bitmapped join indices. *SIGMOD Record* Sep 1995; **24**(3):8–11, doi:10.1145/211990.212001.
7. Kaser O, Lemire D. Threshold and symmetric functions over bitmaps. *Technical Report TR-14-001*, Dept. of CSAS, University of New Brunswick 2014. ArXiv:1402.4073 [cs.DB].

8. Li C, Lu J, Lu Y. Efficient merging and filtering algorithms for approximate string searches. *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE'08, IEEE Computer Society: Washington, DC, USA, 2008; 257–266, doi:10.1109/ICDE.2008.4497434.
9. Sarawagi S, Kirpal A. Efficient set joins on similarity predicates. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD'04, ACM: New York, NY, USA, 2004; 743–754, doi:10.1145/1007568.1007652.
10. Barbay J, Kenyon C. Deterministic algorithm for the t-threshold set problem. *Algorithms and Computation, Lecture Notes in Computer Science*, vol. 2906, Ibaraki T, Katoh N, Ono H (eds.). Springer Berlin Heidelberg, 2003; 575–584, doi:10.1007/978-3-540-24587-2\_59.
11. Lemire D, Kaser O, Aouiche K. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering* 2010; **69**(1):3–28, doi:10.1016/j.datak.2009.08.006.
12. Li M, Jia L, You J, Xi J, Qin H, Zeng R. Fast T-overlap query algorithms using graphics processor units and its applications in web data query. *World Wide Web* 2013; :1–17, doi:10.1007/s11280-013-0232-6.
13. Behm A, Ji S, Li C, Lu J. Space-constrained gram-based indexing for efficient approximate string search. *Proceedings IEEE 25th International Conference on Data Engineering*, ICDE'09, IEEE, 2009; 604–615, doi:10.1109/ICDE.2009.32.
14. Jia L, Xi J, Li M, Liu Y, Miao D. ETI: an efficient index for set similarity queries. *Frontiers of Computer Science* 2012; **6**(6):700–712, doi:10.1007/s11704-012-1237-5.
15. Barbay J, Kenyon C. Adaptive intersection and t-threshold problems. *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'02, Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002; 390–399.
16. Lemire D, Kaser O. Reordering columns for smaller indexes. *Information Sciences* June 2011; **181**(12):2550–2570, doi:10.1016/j.ins.2011.02.002.
17. Navarro G, Provel E. Fast, small, simple rank/select on bitmaps. *Experimental Algorithms, Lecture Notes in Computer Science*, vol. 7276, Klasing R (ed.). Springer Berlin Heidelberg, 2012; 295–306, doi:10.1007/978-3-642-30850-5\_26.
18. Wu K, Stockinger K, Shoshani A. Breaking the curse of cardinality on bitmap indexes. *Scientific and Statistical Database Management, Lecture Notes in Computer Science*, vol. 5069, Ludäscher B, Mamoulis N (eds.). Springer Berlin Heidelberg, 2008; 348–365, doi:10.1007/978-3-540-69497-7\_23.
19. Colantonio A, Di Pietro R. Concise: Compressed 'n' composable integer set. *Information Processing Letters* Jul 2010; **110**(16):644–650, doi:10.1016/j.ipl.2010.05.018.
20. Fusco F, Stoecklin MP, Vlachos M. NET-FLI: On-the-fly compression, archiving and indexing of streaming network traffic. *Proceedings of the VLDB Endowment* 2010; **3**:1382–1393.
21. Guzun G, Canahuate G, Chiu D, Sawin J. A tunable compression framework for bitmap indices. *Proceedings IEEE 30th International Conference on Data Engineering*, ICDE'14, IEEE, 2014.
22. Lemire D, Moon C, McIntosh D, Becho R, Ranger C, Zenz V, Kaser O. JavaEWAH - GitHub page. online: <https://github.com/lemire/javaewah> [13 February 2014].
23. Wu K, Otoo E, Shoshani A. On the performance of bitmap indices for high cardinality attributes. *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB'04, Morgan Kaufmann, 2004; 24–35, doi:10.1016/B978-012088469-8.50006-1.
24. Knuth DE. *Combinatorial Algorithms, Part 1, The Art of Computer Programming*, vol. 4A. Addison-Wesley: Boston, Massachusetts, 2011.
25. Tellez ES, Chávez E, Navarro G. Succinct nearest neighbor search. *Information Systems* 2013; **38**(7):1019–1030, doi:10.1016/j.is.2012.06.005.
26. Critchley A. Finding similar rows in SQL. *Code Project (Webzine)*. CodeProject.com, 2013. Online, <http://www.codeproject.com/Articles/610103/FindingplusSimilarplusRowsplusInplusSQL> [13 February 2014].
27. Ferro A, Giugno R, Puglisi P, Pulvirenti A. An efficient duplicate record detection using q-grams array inverted index. *Data Warehousing and Knowledge Discovery, Lecture Notes in Computer Science*, vol. 6263, Bach Pedersen T, Mohania M, Tjoa A (eds.). Springer Berlin Heidelberg, 2010; 309–323, doi:10.1007/978-3-642-15105-7\_25.
28. Montanari D, Puglisi P. Near duplicate document detection for large information flows. *Multidisciplinary Research and Practice for Information Systems, Lecture Notes in Computer Science*, vol. 7465, Quirchmayr G, Basl J, You I, Xu L, Weippl E (eds.). Springer Berlin Heidelberg, 2012; 203–217, doi:10.1007/978-3-642-32498-7\_16.
29. Katz RH, Borriello G. *Contemporary Logic Design*. 2nd edn., Prentice-Hall, 2004.
30. Garey MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman: New York, 1979.
31. Sethi R. Complete register allocation problems. *SIAM Journal on Computing* 1975; **4**:226–248, doi:10.1137/0204020.
32. Krauthgamer R, Mehta A, Raman V, Rudra A. Greedy list intersection. *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE'08, IEEE Computer Society: Washington, DC, USA, 2008; 1033–1042, doi:10.1109/ICDE.2008.4497512.
33. Ashenden PJ. *Digital Design (Verilog): An Embedded Systems Approach Using Verilog*. Elsevier, 2007.
34. Project Gutenberg Literary Archive Foundation. July 2006 Gutenberg DVD. [http://www.gutenberg.org/wiki/Gutenberg:The\\_CD\\_and\\_DVD\\_Project](http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project) [13 Feb 2014].
35. Frank A, Asuncion A. UCI machine learning repository. <http://archive.ics.uci.edu/ml> [13 February 2014].
36. Engene JO. Five decades of terrorism in Europe: The TWEED dataset. *Journal of Peace Research* 2007; **44**(1):109–121, doi:10.1177/0022343307071497.
37. Webb H, Lemire D, Kaser O. Diamond dicing. *Data & Knowledge Engineering* 2013; **86**:1–18, doi:10.1016/j.datak.2013.01.001.
38. Hahn CJ, Warren SG, London J. Edited synoptic cloud reports from ships and land stations over the globe, 1982–1991. <ftp://cdiac.ornl.gov/pub2/ndp026b/> [13 February 2014].

39. Beyer K, Ramakrishnan R. Bottom-up computation of sparse and iceberg CUBEs. *SIGMOD Record* 1999; **28**(2):359–370, doi:10.1145/304181.304214.
40. Hoos HH. Programming by optimization. *Communications of the ACM* Feb 2012; **55**(2):70–80, doi:10.1145/2076450.2076469.
41. Lagoudakis MG, Littman ML. Algorithm selection using reinforcement learning. *Proceedings of the 17th International Conference on Machine Learning*, ICML'00, Morgan Kaufmann: San Francisco, CA, USA, 2000; 511–518.
42. Horvitz EJ. Reasoning under varying and uncertain resource constraints. *Proceedings of the 7th National Conference on Artificial Intelligence*, AAAI'88, The MIT Press: Cambridge, MA, USA, 1988; 111–116.